REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 2001	Final Report; 7/26/01 - 12/31/01

4. TITLE AND SUBTITLE
Corba Support for Missile Guidance

5. FUNDING NUMBERS
DAAH01-01-C-R160 SubClin # 0002BH

6. AUTHORS Dr. Letha Etzkorn

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
University of Alabama in Huntsville
301 Sparkman Drive
Huntsville, AL 35899

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

U.S. Army Aviation and Missile Command
Redstone Arsenal, AL 35898

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

[redacted]tement A: Unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The tasks that were part of this contract were as follows:

- Teach a short course on CORBA to U.S. Army AMCOM + some contractors
- Develop software demonstrating the CORBA Notification service on the TAO ORB, using an application previously developed by one of the students in the CORBA course as a base for the demonstration.
- Develop a presentation/teach a short course on the CORBA Notification Service.
- Develop a presentation/teach a short course on the Real Time CORBA standard

14. SUBJECT TERMS
CORBA, CORBA Notification Service, Real Time CORBA

15. NUMBER OF PAGES

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	SAR

PLEASE CHECK THE APPROPRIATE BLOCK BELOW

DAO# _____

☐ _____ copies are being forwarded. Indicate whether Statement A, B, C, D, E, F, or X applies.

☑ DISTRIBUTION STATEMENT A:
APPROVED FOR PUBLIC RELEASE: DISTRIBUTION IS UNLIMITED

☐ DISTRIBUTION STATEMENT B:
DISTRIBUTION AUTHORIZED TO U.S. GOVERNMENT AGENCIES ONLY; (indicate Reason and Date). OTHER REQUESTS FOR THIS DOCUMENT SHALL BE REFERRED TO (Indicate Controlling DoD Office).

☐ DISTRIBUTION STATEMENT C:
DISTRIBUTION AUTHORIZED TO U.S. GOVERNMENT AGENCIES AND THEIR CONTRACTS (Indicate Reason and Date). OTHER REQUESTS FOR THIS DOCUMENT SHALL BE REFERRED TO (Indicate Controlling DoD Office).

☐ DISTRIBUTION STATEMENT D:
DISTRIBUTION AUTHORIZED TO DoD AND U.S. DoD CONTRACTORS ONLY; (Indicate Reason and Date). OTHER REQUESTS SHALL BE REFERRED TO (Indicate Controlling DoD Office).

☐ DISTRIBUTION STATEMENT E:
DISTRIBUTION AUTHORIZED TO DoD COMPONENTS ONLY; (Indicate Reason and Date). OTHER REQUESTS SHALL BE REFERRED TO (Indicate Controlling DoD Office).

☐ DISTRIBUTION STATEMENT F:
FUTHER DISSEMINATION ONLY AS DIRECTED BY (Indicate Controlling DoD Office and Date) or HIGHER DoD AUTHORITY.

☐ DISTRIBUTION STATEMENT X:
DISTRIBUTION AUTHORIZED TO U.S. GOVERNMENT AGENCIES AND PRIVATE INDIVIDUALS OR ENTERPRISES ELIGIBLE TO OBTAIN EXPORT-CONTROLLED TECHNICAL DATA IN ACCORDANCE WITH DoD DIRECTIVE 5230.25. WITHHOLDING OF UNCLASSIFIED TECHNICAL DATA FROM PUBLIC DISCLOSURE, 6 Nov 1984 (indicate date of determination). CONTROLLING DoD OFFICE IS (Indicate Controlling DoD Office).

☐ This document was previously forwarded to DTIC on _____ (date) and the AD number is _____.

☐ In accordance with provisions of DoD instructions. The document requested is not supplied because:

☐ It will be published at a later date. (Enter approximate date, if known).

☐ Other. (Give Reason)

DoD Directive 5230.24, "Distribution Statements on Technical Documents," 18 Mar 87, contains seven distribution statements, as described briefly above. Technical Documents must be assigned distribution statements.

Letha Etzkorn
Print or Type Name

Letha Etzkorn
Authorized Signature/Date

(256) 824-6291
Telephone Number

Final Report for

F/DOD/ARMY/AMCOM/CORBA Support for Missile Guidance

Dr. Letha Etzkorn

Computer Science Department

University of Alabama in Huntsville

Several different tasks were performed by the Principal Investigator for Army Contract F/DOD/ARMY/AMCOM/CORBA Support for Missile Guidance, DAAH01-01-C-R16, Option Order 7d. The first task was for the Principal Investigator (Dr. Letha Etzkorn) to teach an 8 day short course on the Common Object Request Broker Architecture (CORBA), which is a middleware standard that is currently receiving wide industrial acceptance, in both the military and commercial realms. This was done at the end of July and early August, 2001. Attendees at the short course included Army civilian employees, and employees from various Army contractors, including OAR Corporation employees. The first 5 days of this short course were taught as all day lectures at an Army facility on Redstone Arsenal. The final 3 days of the short course took place in the UAH Computer Science department laboratories, where students had the opportunity to write programs based on the information they had been previously taught, while being supervised by the Principal Investigator.

The other tasks that were part of this contract were as follows:

- Develop software demonstrating the CORBA Notification service on the TAO ORB, using an application previously developed by one of the students in the CORBA course as a base for the demonstration.
- Develop a presentation/teach a short course on the CORBA Notification Service.
- Develop a presentation/teach a short course on the Real Time CORBA standard

The CORBA Notification service demonstration was developed during September and October, 2001. This required developing several other small CORBA demonstrations, building up to the CORBA Notification service demonstration. The first

demonstration is a demonstration of simple callbacks in CORBA. This is shown in Appendix A of this report. The second demonstration is a demonstration of callbacks using multiple client/servants. This code is shown in Appendix B of this report. Then in Appendix C of this report is a demonstration of the CORBA event service. In Appendix D of this report is shown the final CORBA Notification service demonstration.

Appendix E of this report contains the presentation on the CORBA Notification service, and Appendix F contains the Real Time CORBA presentation. These presentations, and a description of the CORBA Notification service demonstration were presented to civilian Army employees and various contractor employees at the OAR Corporation building, on December 13, 2001.

All of the code shown in these appendices compiles under Microsoft Visual C++ Version 6.0. A separate CDROM containing all of the information in these appendices has already been given to Ms.Amy Ballard, who is the Principal Investigator's point of contact within AMCOM.

In conclusion, all items specified in the Statement of Work have been successfully completed by the Principal Investigator.

Appendix A

Simplest Possible Callback Example

client.cpp

```
#include "QuoterC.h"
#include "d:\TAO_ORB\TAO_zip_version\ACE_wrappers\TAO\orbsvcs\orbsvcs\CosNamingC.h"
#include <iostream>
void show_chunk(const CosNaming::BindingList &bl)
{
        for (CORBA::ULong i=0; i< bl.length(); i++) {
                cout << bl[i].binding_name[0].id;
                if (bl[i].binding_name[0].kind[0] != '\0')
                        cout << "(" << bl[i].binding_name[0].kind << ")";
                if (bl[i].binding_type == CosNaming::ncontext)
                        cout << ": context" << endl;
                else
                        cout << ": reference" << endl;
        }
}
void list_context (CosNaming::NamingContext_ptr nc)
{
        CosNaming::BindingIterator_var it;//Iterator reference
        CosNaming::BindingList_var bl;          //Binding list
        const CORBA::ULong CHUNK=100;                   //Chunk size

        nc->list (CHUNK, bl, it);               //Get first chunk
        show_chunk(bl);                                         //Print first chunk

        if (!CORBA::is_nil(it)) {               //More bindings?
                while (it->next_n(CHUNK,bl))    //Get next chunk
                        show_chunk(bl);                         //Print chunk
                it->destroy();                                  //Clean up
        }
}

int client (int &argc, char** argv)
{
char inchar;
```

```
        try {

    // First initialize the ORB, that will remove some arguments...
    CORBA::ORB_var orb =
     CORBA::ORB_init (argc, argv,
                "myORB0" // the ORB name, it can be anything! );
    CORBA::Object_var naming_context_object =
      orb->resolve_initial_references ("NameService");
    CosNaming::NamingContext_var naming_context =
      CosNaming::NamingContext::_narrow (naming_context_object.in ());

    CosNaming::Name name (1);
    name.length (1);
    name[0].id = CORBA::string_dup ("echo");

    CORBA::Object_var obj =
      naming_context->resolve (name);

    // Now downcast the object reference to the appropriate type
//    Quoter::Stock_Factory_var factory =
//      Quoter::Stock_Factory::_narrow (factory_object.in ());

        echomodule::echo_var myecho;
        try {
                myecho=echomodule::echo::_narrow(obj);
        } catch (const CORBA::SystemException &se) {
                std::cerr << "Cannot narrow reference" << std::endl;
                throw 0;
        }

        // Read the user's input
        std::cout << "n=display Naming context" << std::endl;
        std::cout << "e=call remote echo" << std::endl;
        std::cout << "s=stop" << std::endl;
        std::cout << "Input your command (n, e, or s)" << std::endl;
        std::cin >> inchar;
```

```
                while (inchar != 's') {
                  if (inchar == 'n') {
                        std::cout << "***Naming Context information**********************" <<
std::endl;
                        list_context(naming_context);
                        std::cout << "**************************************************"
<< std:: endl;
                  }
                  else if (inchar == 'e') {
                        std::cout << "Initial message is " << "Harry Potter" << std::endl;
                        CORBA::String_var echostring = myecho->echostring("Harry Potter");
                        std::cout << "Echoed message is " << echostring.in() << std::endl;
                  }

                  // Read the user's input
                  std::cout << "n=display Naming context" << std::endl;
                  std::cout << "e=call remote echo" << std::endl;
                  std::cout << "s=stop" << std::endl;
                  std::cout << "Input your command (n, e, or s)" << std::endl;
                  std::cin >> inchar;
                }
                //NOTE: There's a bug in this code if the ORB is destroyed.  I think it has to do with
                //somehow destroying some _var variables illegally (that's just a guess).  It apparently
                //has NOTHING to do with the data transmission.  This bug was in the original example
                //I downloaded.  It was NOT added by me. I note that apparently none of the code in the
                //Advanced CORBA with C++ book destroys the ORB.
              // Finally destroy the ORB
              //  orb->destroy ();
             }
             catch (CORBA::Exception &) {
              std::cerr << "CORBA exception raised!" << std::endl;
             }
             return 0;
            }
```

ietmmain.cpp

```
#include "QuoterC.h"
#include <windows.h>
#include <stdlib.h>
#include <iostream.h>
#include "ietm.h"

typedef struct {
        int argc;
        char *argv[100];
} thread_params;


// This thread runs the server waiting for callbacks
void ThreadOne(thread_params *args)
{
 server1(args->argc, args->argv);
}


// This thread runs the client
void ThreadTwo(thread_params *args)
{
    client(args->argc, args->argv);
}

// This file starts the threads
int main(int argc, char* argv[])
{
 HANDLE ThreadHandles[2];
 DWORD ThreadOneID, ThreadTwoID;
//  DWORD j;
 int i;
 thread_params tmp;

  // Put the arguments in the struct so can pass to the server code
  tmp.argc=argc;

  i=0;
  while (i<argc) {
        tmp.argv[i]=argv[i];
        i++;
  }

  ThreadHandles[0] = CreateThread( 0,0,
          (LPTHREAD_START_ROUTINE) ThreadOne,
          &tmp, 0, &ThreadOneID);

// Pause a good long while to give the server on the other side time to
// start up before running the client on this side.
//  j = 0;
//        while (i<100) {

          Sleep(10);

//          j++;
```

```
//          }

 // Put the arguments in the struct so can pass to the client code
  tmp.argc=argc;

  i=0;
  while (i<argc) {
          tmp.argv[i]=argv[i];
          i++;
  }

  ThreadHandles[1] = CreateThread( 0,0,
            (LPTHREAD_START_ROUTINE) ThreadTwo,
            &tmp, 0, &ThreadTwoID);


          // Wait for all threads to finish execution
  WaitForMultipleObjects(2, ThreadHandles, TRUE, INFINITE);

//  WaitForMultipleObjects(1, ThreadHandles, TRUE, INFINITE);

           return 0;
}
```

server1.cpp

```
#include "d:\TAO_ORB\TAO_zip_version\ACE_wrappers\TAO\orbsvcs\orbsvcs\CosNamingC.h"
#include <iostream>

int server1(int &argc, char** argv)
{
 try {

   // First initialize the ORB, that will remove some arguments...
   CORBA::ORB_var orb =
    CORBA::ORB_init (argc, argv,
                "myORB1" /* the ORB name, it can be anything! */);

   CORBA::Object_var poa_object =
    orb->resolve_initial_references ("RootPOA");
   PortableServer::POA_var poa =
    PortableServer::POA::_narrow (poa_object.in ());
   PortableServer::POAManager_var poa_manager =
    poa->the_POAManager ();
   poa_manager->activate ();

   // Create the servant
  //  Quoter_Stock_Factory_i stock_factory_i;
          echo_i1 servant;

   // Activate it to obtain the object reference
//    Quoter::Stock_Factory_var stock_factory =
//     stock_factory_i._this ();
          echomodule1::echo1_var myobject=servant._this();

   // Get the Naming Context reference
```

```
  CORBA::Object_var naming_context_object =
    orb->resolve_initial_references ("NameService");
  CosNaming::NamingContext_var naming_context =
    CosNaming::NamingContext::_narrow (naming_context_object.in ());

  // Create and initialize the name.
  CosNaming::Name name (1);
  name.length (1);
  name[0].id = CORBA::string_dup ("echo1");

  // Bind the object
  naming_context->bind (name, myobject.in ());

  orb->run ();

  // Destroy the POA, waiting until the destruction terminates
  poa->destroy (1, 1);
  orb->destroy ();
 }
 catch (CORBA::Exception &) {
  std::cerr << "CORBA exception raised!" << std::endl;
 }
 return 0;
}
```

stock_i1.cpp

```
#include "Stock_i1.h"
#include <iostream>

echo_i1::echo_i1 ()
{
}

char *
echo_i1::echostring1(const char * message) throw (CORBA::SystemException)
{
        std::cout << message << std::endl;

        char * junkie=CORBA::string_dup(message);
        return junkie;
}
```

ietm.h

```
int client (int &argc, char** argv);
int server1(int &argc, char** argv);
```

stock_i1.h

```
#include "Quoter1S.h"
#include <string>

class echo_i1 : public POA_echomodule1::echo1 {
public:
  echo_i1 ();
```

```
 char *echostring1 (const char * message)
        throw (echomodule1::Invalid_Message1);

private:
};

#endif
```

quoter.idl

```
module echomodule
{

 exception Invalid_Message{};

 interface echo
 {
        string echostring(in string message)
                raises (Invalid_Message);
 };
};
```

quoter1.idl

```
module echomodule1
{

 exception Invalid_Message1{};

 interface echo1
 {
        string echostring1(in string message)
                raises (Invalid_Message1);
 };
};
```

client1.cpp

```
#include "Quoter1C.h"
#include "d:\TAO_ORB\TAO_zip_version\ACE_wrappers\TAO\orbsvcs\orbsvcs\CosNamingC.h"
#include <iostream>

void show_chunk(const CosNaming::BindingList &bl)
{
        for (CORBA::ULong i=0; i< bl.length(); i++) {
                cout << bl[i].binding_name[0].id;
                if (bl[i].binding_name[0].kind[0] != '\0')
                        cout << "(" << bl[i].binding_name[0].kind << ")";
                if (bl[i].binding_type == CosNaming::ncontext)
                        cout << ": context" << endl;
                else
                        cout << ": reference" << endl;
        }
}

void list_context (CosNaming::NamingContext_ptr nc)
{
        CosNaming::BindingIterator_var it;//Iterator reference
        CosNaming::BindingList_var bl;          //Binding list
        const CORBA::ULong CHUNK=100;           //Chunk size
```

```
        nc->list (CHUNK, bl, it);                   //Get first chunk
        show_chunk(bl);                             //Print first chunk

        if (!CORBA::is_nil(it)) {                   //More bindings?
                while (it->next_n(CHUNK,bl))        //Get next chunk
                        show_chunk(bl);             //Print chunk
                it->destroy();                      //Clean up
        }
}

int client1 (int &argc, char** argv)
{
int event;
char current_response;
std::cout <<"pooh bear\n";
 try {

  // First initialize the ORB, that will remove some arguments...
  CORBA::ORB_var orb =
   CORBA::ORB_init (argc, argv,
                "myORB1" /* the ORB name, it can be anything! */);

  CORBA::Object_var naming_context_object =
   orb->resolve_initial_references ("NameService");
  CosNaming::NamingContext_var naming_context =
   CosNaming::NamingContext::_narrow (naming_context_object.in ());

  CosNaming::Name name (1);
  name.length (1);
  name[0].id = CORBA::string_dup ("echo1");


  std::cout << "Do you want to send an event?" << std::endl;
  cin >> current_response;

  if (current_response=='n')
                return -1;

  CORBA::Object_var obj =
   naming_context->resolve (name);

//      std::cout << "***Naming Context information**********************" << std::endl;
//      list_context(naming_context);
//      std::cout << "************************************************" << std:: endl;

        echomodule1::echo1_var myecho1;
        try {
                myecho1=echomodule1::echo1::_narrow(obj);
        } catch (const CORBA::SystemException &se) {
                std::cerr << "Cannot narrow reference" << std::endl;
                throw 0;
        }

        std::cout << "Input external event number (-99 to end)";
        std::cout << "Type -99 to end" << std::endl;
```

```
        cin >> event;

        while (event != -99) {


                CORBA::String_var echostring1 = myecho1->echostring1("Ron Weasley");

                char mystring[5];
                char *mystringpointer=mystring;

                _itoa(event, mystringpointer, 10);

                CORBA::String_var echostring2 = myecho1->echostring1(mystringpointer);

                std::cout << "Input external event number (-99 to end)";
                std::cout << "Type -99 to end" << std::endl;
                cin >> event;
        }


        //NOTE: There's a bug in this code if the ORB is destroyed.  I think it has to do with
        //somehow destroying some _var variables illegally (that's just a guess).  It apparently
        //has NOTHING to do with the data transmission.  This bug was in the original example
        //I downloaded.  It was NOT added by me. I note that apparently none of the code in the
        //Advanced CORBA with C++ book destroys the ORB.
  // Finally destroy the ORB
 //  orb->destroy ();
  }
  catch (CORBA::Exception &) {
    std::cerr << "CORBA exception raised!" << std::endl;
  }
  return 0;
}
```

icemain.cpp

```
#include "QuoterC.h"
#include <windows.h>
#include <stdlib.h>
#include <iostream>
#include "ietm1.h"

volatile INT count;
HANDLE semaphore;

typedef struct {
        int argc;
        char *argv[100];
} thread_params;

// This thread handles the server itself
void ThreadOne(thread_params *args)
{

  server(args->argc, args->argv);
}
```

```
void ThreadTwo(thread_params *args)
{
        client1(args->argc, args->argv);
}


// Start up the threads
int main(int argc, char* argv[])
{
 HANDLE ThreadHandles[2];
 DWORD ThreadOneID, ThreadTwoID;
//  DWORD j;
 int i;
 thread_params tmp;


  // Put the arguments in the struct so can pass to the server code
  tmp.argc=argc;

  i=0;
  while (i<argc) {
        tmp.argv[i]=argv[i];
        i++;
  }


  ThreadHandles[0] = CreateThread( 0,0,
          (LPTHREAD_START_ROUTINE) ThreadOne,
          &tmp, 0, &ThreadOneID);

        // Pause a good long while to give the server on the other side time to
   // start up before running the client on this side.
//   j = 0;
//        while (i<100) {

          Sleep(10);

//          j++;

//        }

 // Put the arguments in the struct so can pass to the client code
 tmp.argc=argc;

  i=0;
  while (i<argc) {
        tmp.argv[i]=argv[i];
        i++;
  }

  ThreadHandles[1] = CreateThread( 0,0,
          (LPTHREAD_START_ROUTINE) ThreadTwo,
          &tmp, 0, &ThreadTwoID);


        // Wait for all threads to finish execution
```

```
  WaitForMultipleObjects(2, ThreadHandles, TRUE, INFINITE);

//  WaitForMultipleObjects(1, ThreadHandles, TRUE, INFINITE);

  CloseHandle(semaphore);

        return 0;
}
```

server.cpp

```
#include "Stock_i.h"
#include "d:\TAO_ORB\TAO_zip_version\ACE_wrappers\TAO\orbsvcs\orbsvcs\CosNamingC.h"
#include <iostream>

int server (int &argc, char** argv)
{
 try {
  // First initialize the ORB, that will remove some arguments...
  CORBA::ORB_var orb =
   CORBA::ORB_init (argc, argv,
            "myORB0" /* the ORB name, it can be anything! */);
  CORBA::Object_var poa_object =
   orb->resolve_initial_references ("RootPOA");
  PortableServer::POA_var poa =
   PortableServer::POA::_narrow (poa_object.in ());
  PortableServer::POAManager_var poa_manager =
   poa->the_POAManager ();
  poa_manager->activate ();

  // Create the servant
 //  Quoter_Stock_Factory_i stock_factory_i;
        echo_i servant;

  // Activate it to obtain the object reference
//   Quoter::Stock_Factory_var stock_factory =
//    stock_factory_i._this ();
        echomodule::echo_var myobject=servant._this();

  // Get the Naming Context reference
  CORBA::Object_var naming_context_object =
   orb->resolve_initial_references ("NameService");
  CosNaming::NamingContext_var naming_context =
   CosNaming::NamingContext::_narrow (naming_context_object.in ());

  // Create and initialize the name.
  CosNaming::Name name (1);
  name.length (1);
  name[0].id = CORBA::string_dup ("echo");

  // Bind the object
  naming_context->bind (name, myobject.in ());

  orb->run ();

  // Destroy the POA, waiting until the destruction terminates
  poa->destroy (1, 1);
```

```
   orb->destroy ();
  }
  catch (CORBA::Exception &) {
   std::cerr << "CORBA exception raised!" << std::endl;
  }
  return 0;
}
```

stock_i.cpp

```
#include "Stock_i.h"
#include <iostream>
#include "ietm1.h"

echo_i::echo_i ()

{
}


char *
echo_i::echostring(const char * message) throw (CORBA::SystemException)
{
        std::cout << message << std::endl;

        char * junkie=CORBA::string_dup(message);
        return junkie;
}
```

ietm.h

```
int server (int &argc, char** argv);
int client1(int &argc, char** argv);
```

stock_i.h

```
#include "QuoterS.h"
#include <string>

class echo_i : public POA_echomodule::echo {
public:
  echo_i ();

  char *echostring (const char * message)
          throw (echomodule::Invalid_Message);

private:
};
```

Appendix B
Callback Demonstration Using Multiple Client/Servants

client.cpp

```
include "QuoterC.h"
#include "Stock_i1.h"
#include "d:\TAO_ORB\TAO_zip_version\ACE_wrappers\TAO\orbsvcs\orbsvcs\CosNamingC.h"
#include <iostream>
#include "ietm.h"


// Object references of servants
extern CORBA::String_var servant_IORs[max_num_servants];
extern echo_i1 servant[max_num_servants];
extern int current_num_servants;


void show_chunk(const CosNaming::BindingList &bl)
{
        for (CORBA::ULong i=0; i< bl.length(); i++) {
                cout << bl[i].binding_name[0].id;
                if (bl[i].binding_name[0].kind[0] != '\0')
                        cout << "(" << bl[i].binding_name[0].kind << ")";
                if (bl[i].binding_type == CosNaming::ncontext)
                        cout << ": context" << endl;
                else
                        cout << ": reference" << endl;
        }
}

void list_context (CosNaming::NamingContext_ptr nc)
{
        CosNaming::BindingIterator_var it;//Iterator reference
        CosNaming::BindingList_var bl;          //Binding list
        const CORBA::ULong CHUNK=100;                   //Chunk size

        nc->list (CHUNK, bl, it);               //Get first chunk
        show_chunk(bl);                                 //Print first chunk

        if (!CORBA::is_nil(it)) {               //More bindings?
                while (it->next_n(CHUNK,bl))    //Get next chunk
                        show_chunk(bl);                         //Print chunk
                it->destroy();                                  //Clean up
        }
}


int client (int &argc, char** argv)
{
char inchar;

        try {

  // First initialize the ORB, that will remove some arguments...
  CORBA::ORB_var orb =
```

```
    CORBA::ORB_init (argc, argv,
                "myORB0" // the ORB name, it can be anything!
                                                );

    CORBA::Object_var naming_context_object =
      orb->resolve_initial_references ("NameService");
    CosNaming::NamingContext_var naming_context =
      CosNaming::NamingContext::_narrow (naming_context_object.in ());

    CosNaming::Name name (1);
    name.length (1);
    name[0].id = CORBA::string_dup ("echo");

    CORBA::Object_var obj =
      naming_context->resolve (name);

    // Now downcast the object reference to the appropriate type
//    Quoter::Stock_Factory_var factory =
//      Quoter::Stock_Factory::_narrow (factory_object.in ());

          echomodule::echo_var myecho;
          try {
                  myecho=echomodule::echo::_narrow(obj);
          } catch (const CORBA::SystemException &se) {
                  std::cerr << "Cannot narrow reference" << std::endl;
                  throw 0;
          }

          // Read the user's input
          std::cout << "n=display Naming context" << std::endl;
          std::cout << "e=call remote echo" << std::endl;
          std::cout << "r=register callback" << std::endl;
          std::cout << "s=stop" << std::endl;
          std::cout << "u=unregister callback" << std::endl;
          std::cout << "Input your command (n, e, r, or s)" << std::endl;
          std::cin >> inchar;

          while (inchar != 's') {
            if (inchar == 'n') {
                    std::cout << "***Naming Context information**********************" << std::endl;
                    list_context(naming_context);
                    std::cout << "**************************************************" << std::
endl;
            }
            else if (inchar == 'e') {
                    std::cout << "Initial message is " << "Harry Potter" << std::endl;
                    CORBA::String_var echostring = myecho->echostring("Harry Potter");
                    std::cout << "Echoed message is " << echostring.in() << std::endl;
            }
            else if (inchar == 'r') {
                     int inval;

                     std::cout << "About to register a callback\n";

                     inval=-1;
                     while ((inval < 0) || (inval > current_num_servants) ) {
```

```
                std::cout << "Which clientservant do you want to register? (numeric value)\n";
                cin >> inval;
                        if ((inval < 0) || (inval > current_num_servants) )
                                std::cout << "---Illegal clientservant number, try again\n";
                }


                myecho->register_callback(servant_IORs[inval],servant[inval].myName);
                std::cout << "Just finished registering a callback\n";
        }
        else if (inchar == 'u') {
                int inval;

                std::cout << "About to unregister a callback\n";

                inval=-1;
                while ((inval < 0) || (inval > current_num_servants) ) {
                 std::cout << "Which clientservant do you want to unregister? (numeric value)\n";
                 cin >> inval;
                        if ((inval < 0) || (inval > current_num_servants) )
                                std::cout << "---Illegal clientservant number, try again\n";
                }


                myecho->unregister_callback(servant_IORs[inval],servant[inval].myName);
                std::cout << "Just finished unregistering a callback\n";
        }


        // Read the user's input
        std::cout << "n=display Naming context" << std::endl;
        std::cout << "e=call remote echo" << std::endl;
        std::cout << "r=register callback" << std::endl;
        std::cout << "u=unregister callback" << std::endl;
        std::cout << "s=stop" << std::endl;
        std::cout << "Input your command (n, e, r, or s)" << std::endl;
        std::cin >> inchar;
       }

      //NOTE: There's a bug in this version of the TAO ORB if the ORB is destroyed.
  // Finally destroy the ORB
  //  orb->destroy ();

 }
 catch (CORBA::Exception &) {
  std::cerr << "CORBA exception raised!" << std::endl;
 }

 return 0;
}
```

jetmmain.cpp

```
#include "QuoterC.h"
#include <windows.h>
#include <stdlib.h>
#include <iostream.h>
```

```
#include "ietm.h"

typedef struct {
        int argc;
        char *argv[100];
} thread_params;


// This thread runs the server waiting for callbacks
void ThreadOne(thread_params *args)
{
 server1(args->argc, args->argv);
}

// This thread runs the client
void ThreadTwo(thread_params *args)
{

    client(args->argc, args->argv);

}

// This file starts the threads
int main(int argc, char* argv[])
{
 HANDLE ThreadHandles[2];
 DWORD ThreadOneID, ThreadTwoID;
//  DWORD j;
 int i;
 thread_params tmp;

  // Put the arguments in the struct so can pass to the server code
  tmp.argc=argc;

  i=0;
  while (i<argc) {
        tmp.argv[i]=argv[i];
        i++;
  }

  ThreadHandles[0] = CreateThread( 0,0,
          (LPTHREAD_START_ROUTINE) ThreadOne,
          &tmp, 0, &ThreadOneID);

        // Pause a good long while to give the server on the other side time to
  // start up before running the client on this side.
//  j = 0;
//        while (i<100) {

          Sleep(10);

//          j++;

//        }
```

```
// Put the arguments in the struct so can pass to the client code
 tmp.argc=argc;

 i=0;
 while (i<argc) {
        tmp.argv[i]=argv[i];
        i++;
 }

 ThreadHandles[1] = CreateThread( 0,0,
          (LPTHREAD_START_ROUTINE) ThreadTwo,
          &tmp, 0, &ThreadTwoID);


        // Wait for all threads to finish execution
 WaitForMultipleObjects(2, ThreadHandles, TRUE, INFINITE);

//  WaitForMultipleObjects(1, ThreadHandles, TRUE, INFINITE);

         return 0;
}
```

server1.cpp

```
#include "Stock_i1.h"
#include "d:\TAO_ORB\TAO_zip_version\ACE_wrappers\TAO\orbsvcs\orbsvcs\CosNamingC.h"
#include <iostream>
#include "ietm.h"


// Current number of servants
int current_num_servants = 0;
// Object references of servants
CORBA::String_var servant_IORs[max_num_servants];
echo_i1 servant[max_num_servants];

int server1(int &argc, char** argv)
{
 try {


  // First initialize the ORB, that will remove some arguments...
  CORBA::ORB_var orb =
   CORBA::ORB_init (argc, argv,
            "myORB1" /* the ORB name, it can be anything! */);


  CORBA::Object_var poa_object =
   orb->resolve_initial_references ("RootPOA");
  PortableServer::POA_var poa =
   PortableServer::POA::_narrow (poa_object.in ());
  PortableServer::POAManager_var poa_manager =
   poa->the_POAManager ();
  poa_manager->activate ();

  // Create the servants
/*        echo_i1 servant[max_num_servants] =
```

```
{echo_i1(0), echo_i1(1), echo_i1(2), echo_i1(3), echo_i1(4) }; */

// Activate a servant to obtain an object reference
        echomodule1::echo1_var myobject;

        myobject = servant[0]._this();
        servant_IORs[0] = orb->object_to_string(myobject.in());
        servant[0].myServantNumber=0;
        strcpy(servant[0].myName,"A0");
        current_num_servants = 0;

// Activate a servant to obtain an object reference
        myobject = servant[1]._this();
        servant_IORs[1] = orb->object_to_string(myobject.in());
        servant[1].myServantNumber=1;
        strcpy(servant[1].myName,"A1");
        current_num_servants = 1;

          // Activate a servant to obtain an object reference
        myobject = servant[2]._this();
        servant_IORs[2] = orb->object_to_string(myobject.in());
        servant[2].myServantNumber=2;
        strcpy(servant[2].myName,"A2");
        current_num_servants = 2;

          // Activate a servant to obtain an object reference
        myobject = servant[3]._this();
        servant_IORs[3] = orb->object_to_string(myobject.in());
        servant[3].myServantNumber=3;
        strcpy(servant[3].myName,"A3");
        current_num_servants = 3;

          // Activate a servant to obtain an object reference
        myobject = servant[4]._this();
        servant_IORs[4] = orb->object_to_string(myobject.in());
        servant[4].myServantNumber=4;
        strcpy(servant[4].myName,"A4");
        current_num_servants = 4;


/*  // Get the Naming Context reference
   CORBA::Object_var naming_context_object =
     orb->resolve_initial_references ("NameService");
   CosNaming::NamingContext_var naming_context =
     CosNaming::NamingContext::_narrow (naming_context_object.in ());

   // Create and initialize the name.
   CosNaming::Name name (1);
   name.length (1);
   name[0].id = CORBA::string_dup ("echo1");

   // Bind the object
   naming_context->bind (name, myobject.in ()); */
```

```
  orb->run ();

  // Destroy the POA, waiting until the destruction terminates
  poa->destroy (1, 1);
  orb->destroy ();
 }
 catch (CORBA::Exception &) {
  std::cerr << "CORBA exception raised!" << std::endl;
 }
 return 0;
}
```

stock_i1.cpp

```
#include "Stock_i1.h"
#include <iostream>

echo_i1::echo_i1 ()
{
        myServantNumber=-1;
}

/*echo_i1::echo_i1 (int num)
{
        myServantNumber=num;
} */

char *
echo_i1::event_notification(const char * message) throw (CORBA::SystemException)
{
        std::cout << "Clientserver " << myServantNumber << " received: ";
        std::cout << "  " << message << std::endl;

        char * junkie=CORBA::string_dup("Event has been received\n");
cout << "About to return from event notification\n";
        return junkie;
}
```

ietm.h

```
#define max_num_servants 5
int client (int &argc, char** argv);
int server1(int &argc, char** argv);
```

stock_i1.h

```
#include "Quoter1S.h"
#include <string>

class echo_i1 : public POA_echomodule1::echo1 {
public:
  echo_i1();
  echo_i1 (int num);

  char *event_notification (const char * message)
          throw (echomodule1::Invalid_Message1);

  int myServantNumber;
  char myName[50];
```

```
private:

};
#endif
```

quoter.idl

```
module echomodule
{

 exception Invalid_Message{ };

 interface echo
 {
        string echostring(in string message)
                raises (Invalid_Message);
        void register_callback(in string message, in string myname)
                raises (Invalid_Message);
        void unregister_callback(in string message, in string myname)
                raises (Invalid_Message);

 };
};
```

quoter1.idl

```
module echomodule1
{

 exception Invalid_Message1{ };

 interface echo1
 {
        string event_notification(in string message)
                raises (Invalid_Message1);
 };
};
```

client1.cpp

```
#include "Quoter1C.h"
#include <windows.h>
#include "d:\TAO_ORB\TAO_zip_version\ACE_wrappers\TAO\orbsvcs\orbsvcs\CosNamingC.h"
#include <iostream>
#include "ietm1.h"
#include "register.h"

extern int event;
extern HANDLE event_semaphore;
extern HANDLE register_callback_semaphore;

CORBA::ORB_var orb;


int clientservant_count = -1; // Current number of clientservants
extern registered_clientservant ClientServant[max_num_clientservants];


void show_chunk(const CosNaming::BindingList &bl)
```

```
{
        for (CORBA::ULong i=0; i< bl.length(); i++) {
                cout << bl[i].binding_name[0].id;
                if (bl[i].binding_name[0].kind[0] != '\0')
                        cout << "(" << bl[i].binding_name[0].kind << ")";
                if (bl[i].binding_type == CosNaming::ncontext)
                        cout << ": context" << endl;
                else
                        cout << ": reference" << endl;
        }
}

void list_context (CosNaming::NamingContext_ptr nc)
{
        CosNaming::BindingIterator_var it;//Iterator reference
        CosNaming::BindingList_var bl;          //Binding list
        const CORBA::ULong CHUNK=100;                   //Chunk size

        nc->list (CHUNK, bl, it);               //Get first chunk
        show_chunk(bl);                                 //Print first chunk

        if (!CORBA::is_nil(it)) {               //More bindings?
                while (it->next_n(CHUNK,bl))    //Get next chunk
                        show_chunk(bl);                 //Print chunk
                it->destroy();                          //Clean up
        }
}


int client1 (int &argc, char** argv)
{

int i;

 try {

          // First initialize the ORB, that will remove some arguments...
  orb =
    CORBA::ORB_init (argc, argv,
                "myORB1" /* the ORB name, it can be anything! */);

        // Wait until at least one callback is registered
        WaitForSingleObject(register_callback_semaphore,INFINITE);



        // Send the event to all registered clientservants

        while (event != -99) {

          // Wait until an event occurs
          WaitForSingleObject(event_semaphore,INFINITE);

          i=0;
//cout << "About to enter event sending loop\n";
          while (i < max_num_clientservants) {
```

```
//cout << "About to check if ClientServant[" << i << "] registered\n";
                        if (ClientServant[i].registered_or_not !=0) {
                                // Convert the event number to a string for transmission
                                char mystring[5];
                                char *mystringpointer=mystring;

                                _itoa(event, mystringpointer, 10);

//cout << "About to call ClientServant[" << i << "]\n";
                                CORBA::String_var event_return2 = ClientServant[i].myecho-
>event_notification(ClientServant[i].myname);
                                CORBA::String_var event_return3 = ClientServant[i].myecho-
>event_notification(mystringpointer);
//cout << "Just finished calling ClientServant[" << i << "]\n";
                        }
//cout << "About to increment i\n";
                    i++;

            }
          }


          //NOTE: There's a bug in this code if the ORB is destroyed.  I think it has to do with
          //somehow destroying some _var variables illegally (that's just a guess).  It apparently
          //has NOTHING to do with the data transmission.  This bug was in the original example
          //I downloaded.  It was NOT added by me. I note that apparently none of the code in the
          //Advanced CORBA with C++ book destroys the ORB.
    // Finally destroy the ORB
   //  orb->destroy ();
   }
   catch (CORBA::Exception &) {
     std::cerr << "CORBA exception raised!" << std::endl;
   }
   return 0;
}
```

icemain.cpp

```
#include "QuoterC.h"
#include "Quoter1C.h"
#include <windows.h>
#include <stdlib.h>
#include <iostream>
#include "ietm1.h"
#include "register.h"

volatile INT count;

int event;                      // Current event to send to remote
HANDLE event_semaphore;
HANDLE register_callback_semaphore;
HANDLE register_callback_semaphore1;
registered_clientservant ClientServant[max_num_clientservants];

void initialize_ClientServantarray() {
int i;
```

```
 i=0;
 while (i < max_num_clientservants) {

          ClientServant[i].registered_or_not=0;
          i++;
 }
}

void ReleaseEventSemaphore() {
  LONG semaCount;

  ReleaseSemaphore(event_semaphore, 1, &semaCount);
}

void ReleaseRegisterCallbackSemaphore() {
  LONG semaCount;

  ReleaseSemaphore(register_callback_semaphore, 1, &semaCount);
}
void ReleaseRegisterCallbackSemaphore1() {
  LONG semaCount;
;
  ReleaseSemaphore(register_callback_semaphore1, 1, &semaCount);
}


typedef struct {
          int argc;
          char *argv[100];
} thread_params;


// This thread handles the server itself
void ThreadOne(thread_params *args)
{

 server(args->argc, args->argv);
}


// ThreadTwo handles the client used for callbacks
void ThreadTwo(thread_params *args)
{

          client1(args->argc, args->argv);
}

// This thread simulates input events
void ThreadThree(thread_params *args)
{

          // Wait until a callback is registered before prompting for event input
          WaitForSingleObject(register_callback_semaphore1,INFINITE);

          while (1) {  // Do forever
```

```
            std::cout << "Input an event # to send to the remote" << std::endl;
            cin >> event;
            ReleaseEventSemaphore();

          }

}


// Start up the threads
int main(int argc, char* argv[])
{
 HANDLE ThreadHandles[3];
 DWORD ThreadOneID, ThreadTwoID, ThreadThreeID;
//  DWORD j;
 int i;
 thread_params tmp;


 initialize_ClientServantarray();


  // Put the arguments in the struct so can pass to the server code
  tmp.argc=argc;

  i=0;
  while (i<argc) {
          tmp.argv[i]=argv[i];
          i++;
  }


  ThreadHandles[0] = CreateThread( 0,0,
            (LPTHREAD_START_ROUTINE) ThreadOne,
            &tmp, 0, &ThreadOneID);



         // Pause a good long while to give the server on the other side time to
   // start up before running the client on this side.
//   j = 0;
//          while (i<100) {

            Sleep(10);

//          j++;

//          }


 // Put the arguments in the struct so can pass to the client code
  tmp.argc=argc;

  i=0;
  while (i<argc) {
          tmp.argv[i]=argv[i];
```

```
        i++;
  }


        // Start the semaphores off at zero,in order to initially block the
        // local client from sending events to the remote.  Register callback
  // is initially turned off, and so is the current event
  register_callback_semaphore = CreateSemaphore(0,0,1,0);
  register_callback_semaphore1 = CreateSemaphore(0,0,1,0);
  event_semaphore = CreateSemaphore(0,0,1,0);

  ThreadHandles[1] = CreateThread( 0,0,
          (LPTHREAD_START_ROUTINE) ThreadTwo,
          &tmp, 0, &ThreadTwoID);

// Put the arguments in the struct so can pass to the client code
  tmp.argc=argc;

  i=0;
  while (i<argc) {
        tmp.argv[i]=argv[i];
        i++;
  }

  ThreadHandles[2] = CreateThread( 0,0,
          (LPTHREAD_START_ROUTINE) ThreadThree,
          &tmp, 0, &ThreadThreeID);

        // Wait for all threads to finish execution
  WaitForMultipleObjects(2, ThreadHandles, TRUE, INFINITE);

//  WaitForMultipleObjects(1, ThreadHandles, TRUE, INFINITE);

  CloseHandle(event_semaphore);

        return 0;
}
```

server.cpp

```
#include "Stock_i.h"
#include "d:\TAO_ORB\TAO_zip_version\ACE_wrappers\TAO\orbsvcs\orbsvcs\CosNamingC.h"
#include <iostream>

int server (int &argc, char** argv)
{
 try {
  // First initialize the ORB, that will remove some arguments...
  CORBA::ORB_var orb =
   CORBA::ORB_init (argc, argv,
          "myORB0" /* the ORB name, it can be anything! */);
  CORBA::Object_var poa_object =
   orb->resolve_initial_references ("RootPOA");
  PortableServer::POA_var poa =
   PortableServer::POA::_narrow (poa_object.in ());
  PortableServer::POAManager_var poa_manager =
   poa->the_POAManager ();
```

```
    poa_manager->activate ();

    // Create the servant
   // Quoter_Stock_Factory_i stock_factory_i;
          echo_i servant;

    // Activate it to obtain the object reference
//    Quoter::Stock_Factory_var stock_factory =
//      stock_factory_i._this ();
          echomodule::echo_var myobject=servant._this();

    // Get the Naming Context reference
    CORBA::Object_var naming_context_object =
     orb->resolve_initial_references ("NameService");
    CosNaming::NamingContext_var naming_context =
     CosNaming::NamingContext::_narrow (naming_context_object.in ());

    // Create and initialize the name.
    CosNaming::Name name (1);
    name.length (1);
    name[0].id = CORBA::string_dup ("echo");

    // Bind the object
    naming_context->bind (name, myobject.in ());

    orb->run ();

    // Destroy the POA, waiting until the destruction terminates
    poa->destroy (1, 1);
    orb->destroy ();
   }
   catch (CORBA::Exception &) {
    std::cerr << "CORBA exception raised!" << std::endl;
   }
   return 0;
}
```

stock_i.cpp

```
#include "Quoter1C.h"
#include "Stock_i.h"
#include <iostream>
#include <string.h>
#include "ietm1.h"
#include "register.h"

extern int clientservant_count;  // Current number of clientservants

//  Proxy objects for clientservants
extern registered_clientservant ClientServant[max_num_clientservants];
extern CORBA::ORB_var orb;


// Search for the lowest numbered free ClientServer entry in the ClientServer array
int return_free_ClientServant_entry (){
int i;
```

```
        i=0;
        while (i < max_num_clientservants) {
                if (ClientServant[i].registered_or_not == 0) {
                        ClientServant[i].registered_or_not = 1;
                        return i;
                }

                i++;
        }

        return -1;

}


int find_ClientServant_entry(const char * myname) {

int i;

        char * junkie=CORBA::string_dup(myname);

//std::cout << "Finding entry to unregister\n";

        i=0;
        while (i < max_num_clientservants) {
                // Only check those entries that are actually registered
                if (ClientServant[i].registered_or_not == 1) {
//std::cout << "Looking at registered entries\n";
                        // Look for an entry with the same Name
                        if (!strcmp(ClientServant[i].myname.in(),
                                junkie)) {
//std::cout << "Found an entry with the same IOR\n";
//std::cout << "i is " << i << "\n";
//std::cout << "IOR is " << ClientServant[i].myIOR.in();
                                return i;
                        }
                }
//std::cout << "Incrementing i\n";
                i++;
        }
//std::cout << "Returning not found\n";
        return -1;

}


echo_i::echo_i ()

{
}


char *
echo_i::echostring(const char * message) throw (CORBA::SystemException)
{
        std::cout << message << std::endl;
```

```
        char * junkie=CORBA::string_dup(message);
        return junkie;
}

void echo_i::register_callback(const char * message, const char * myname) throw
(CORBA::SystemException) {

int free_ClientServantentry;

//std::cout << "Registering a callback" << std::endl;
        ReleaseRegisterCallbackSemaphore();
        ReleaseRegisterCallbackSemaphore1();

//std::cout << "clientservant_count = " << clientservant_count << "\n";

        if (clientservant_count < max_num_clientservants)  {



                free_ClientServantentry= return_free_ClientServant_entry();

                if (free_ClientServantentry <0) {
                        std::cout << "No more ClientServants allowed\n";
                        exit (1);
                }
//std::cout << "free_ClientServantentry = " << free_ClientServantentry << "\n";

                clientservant_count++;

//std::cout << "About to duplicate message0\n";
                char * junkie=CORBA::string_dup(message);
//std::cout << "About to duplicate message1\n";
//std::cout << "free_ClientServantentry = " << free_ClientServantentry << "\n";

                ClientServant[free_ClientServantentry].myIOR = CORBA::string_dup(junkie);
//CORBA::String_var jj=CORBA::string_dup(junkie);
//std::cout << "Just stored IOR\n";
                CORBA::Object_var obj = orb->string_to_object(message);
//std::cout << "Just converted IOR to object\n";
                if (CORBA::is_nil(obj)) {
                        std::cerr << "Nil reference" << std:: endl;
                        throw 0;
                }

                char *junkie1=CORBA::string_dup(myname);
                ClientServant[free_ClientServantentry].myname = CORBA::string_dup(junkie1);

                try {

        ClientServant[free_ClientServantentry].myecho=echomodule1::echo1::_narrow(obj);
//std::cout << "Just narrowed IOR\n";
                }       catch (const CORBA::SystemException &se) {
                        std::cerr << "Cannot narrow reference" << std::endl;
                        throw 0;
```

```
                }

        }

//std::cout << "A callback was just registered\n";

}

void echo_i::unregister_callback(const char * message, const char * myname) throw
(CORBA::SystemException) {

int i;

std::cout << "Unregistering\n";

        i=find_ClientServant_entry(myname);
        ClientServant[i].registered_or_not = 0;
}
```

ietm1.h

```
#define max_num_clientservants 5
int server (int &argc, char** argv);
int client1(int &argc, char** argv);
void ReleaseEventSemaphore();
void ReleaseRegisterCallbackSemaphore();
void ReleaseRegisterCallbackSemaphore1();
```

register.h

```
struct registered_clientservant {

        echomodule1::echo1_var myecho; //  Proxy objects for clientservants

        CORBA::String_var myIOR;  // IORs associated with proxy objects

        CORBA::String_var myname;

        int registered_or_not;

};
```

stock_i.h

```
#include "QuoterS.h"
#include <string>

class echo_i : public POA_echomodule::echo {
public:
  echo_i ();

  char *echostring (const char * message)
          throw (echomodule::Invalid_Message);
  void register_callback(const char * message, const char *myname)
          throw (echomodule::Invalid_Message);
  void unregister_callback(const char * message, const char *myname)
          throw (echomodule::Invalid_Message);
```

```
private:
};
```

Appendix C
CORBA Event Service Demonstration

client.cpp

```
#include "QuoterC.h"
#include "d:\TAO_ORB\TAO_zip_version\ACE_wrappers\TAO\orbsvcs\orbsvcs\CosNamingC.h"
#include <iostream>

extern CORBA::ORB_var orb;


void show_chunk(const CosNaming::BindingList &bl)
{
        for (CORBA::ULong i=0; i< bl.length(); i++) {
                cout << bl[i].binding_name[0].id;
                if (bl[i].binding_name[0].kind[0] != '\0')
                        cout << "(" << bl[i].binding_name[0].kind << ")";
                if (bl[i].binding_type == CosNaming::ncontext)
                        cout << ": context" << endl;
                else
                        cout << ": reference" << endl;
        }
}

void list_context (CosNaming::NamingContext_ptr nc)
{
        CosNaming::BindingIterator_var it;//Iterator reference
        CosNaming::BindingList_var bl;          //Binding list
        const CORBA::ULong CHUNK=100;                   //Chunk size

        nc->list (CHUNK, bl, it);               //Get first chunk
        show_chunk(bl);                                 //Print first chunk

        if (!CORBA::is_nil(it)) {               //More bindings?
                while (it->next_n(CHUNK,bl))    //Get next chunk
                        show_chunk(bl);                 //Print chunk
                it->destroy();                          //Clean up
        }
}


int client (int &argc, char** argv)
{
char inchar;

std::cout << "Top of client\n";
        try {

                CORBA::Object_var naming_context_object;
std::cout << "Client, inside try\n";

                try {
                naming_context_object=
            orb->string_to_object (argv[2]);
                } catch (const CORBA::SystemException &se) {
```

```
                        std::cout << "Cannot resolve NameService address" << std::endl;
                        throw 0;
                }
//   CORBA::Object_var naming_context_object =
//     orb->resolve_initial_references ("NameService");
std::cout << "Just past resolve NameService address\n";


                CosNaming::NamingContext_var naming_context;

                try {
                naming_context =
                        CosNaming::NamingContext::_narrow (naming_context_object.in ());
                } catch (const CORBA::SystemException &se) {
                        std::cout << "Cannot narrow naming_context_object" << std::endl;
                        throw 0;
                }

std::cout << "Just past narrow NameService address\n";

        CosNaming::Name name (1);
        name.length (1);
        name[0].id = CORBA::string_dup ("echo");

        CORBA::Object_var obj;

        try {
                        obj =
                                naming_context->resolve (name);
                } catch (const CORBA::SystemException &se) {
                        std::cout << "Cannot resolve name" << std::endl;
                        throw 0;
                }
std::cout << "Just past resolve name address\n";

   // Now downcast the object reference to the appropriate type
//   Quoter::Stock_Factory_var factory =
//     Quoter::Stock_Factory::_narrow (factory_object.in ());

                echomodule::echo_var myecho;
                try {
                        myecho=echomodule::echo::_narrow(obj);
                } catch (const CORBA::SystemException &se) {
                        std::cerr << "Cannot narrow reference" << std::endl;
                        throw 0;
                }
std::cout << "Just done narrowing echo\n";
        // Read the user's input
                std::cout << "n=display Naming context" << std::endl;
                std::cout << "e=call remote echo" << std::endl;
                std::cout << "r=register callback" << std::endl;
                std::cout << "s=stop" << std::endl;
                std::cout << "Input your command (n, e, r, or s)" << std::endl;
                std::cin >> inchar;

                while (inchar != 's') {
```

```
                    if (inchar == 'n') {
                            std::cout << "***Naming Context information**********************" <<
std::endl;
                            list_context(naming_context);
                            std::cout << "*****************************************************"
<< std:: endl;
                    }
                    else if (inchar == 'e') {
                            std::cout << "Initial message is " << "Harry Potter" << std::endl;
                            CORBA::String_var echostring = myecho->echostring("Harry Potter");
                            std::cout << "Echoed message is " << echostring.in() << std::endl;
                    }
                    else if (inchar == 'r') {
                             CORBA::Short i=myecho->register_callback("clientA");
                    }

                   // Read the user's input
                   std::cout << "n=display Naming context" << std::endl;
                   std::cout << "e=call remote echo" << std::endl;
                   std::cout << "r=register callback" << std::endl;
                   std::cout << "s=stop" << std::endl;
                   std::cout << "Input your command (n, e, r, or s)" << std::endl;
                    std::cin >> inchar;
                   }

          //NOTE: There's a bug in this code if the ORB is destroyed.  I think it has to do with
          //somehow destroying some _var variables illegally (that's just a guess).  It apparently
          //has NOTHING to do with the data transmission.  This bug was in the original example
          //I downloaded.  It was NOT added by me. I note that apparently none of the code in the
          //Adanced CORBA with C++ book destroys the ORB.
   // Finally destroy the ORB
   //  orb->destroy ();

          }
          catch (CORBA::Exception &) {
                  std::cerr << "CORBA exception raised!" << std::endl;
          }

 return 0;
}
```

consumer.cpp

```
#include "Consumer.h"
//#include "orbsvcs/CosEventChannelAdminS.h"
#include
"d:\TAO_ORB\TAO_zip_version\ACE_wrappers\TAO\orbsvcs\orbsvcs\CosEventChannelAdminS.h"
#include <iostream>
using namespace std;

extern int thread_main(int argc, char* argv[]);

CORBA::ORB_var orb;

// *********************************************************************

Consumer::Consumer (void)
```

```
 : event_count_ (0)
{
}


void
Consumer::push (const CORBA::Any & current_event,
                CORBA::Environment &)
{
 CORBA::ULong myevent;

 // Extract Any current_event, check its type for
 // validity
 if (! (current_event >>= myevent) ) {
        std::cout << "event was not of type CORBA::ULong\n";
 }
 else {
        // Write out event number
        std::cout << "Event number is " << myevent << "\n";
 }

//  this->event_count_ ++;
//  if (this->event_count_ % 100 == 0)
//    {
//        std::cout << " Received event #" <<
//                this->event_count_ << std::endl;
//    }

}

void
Consumer::disconnect_push_consumer (CORBA::Environment &ACE_TRY_ENV)
{
 // In this example we shutdown the ORB when we disconnect from the
 // EC (or rather the EC disconnects from us), but this doesn't have
 // to be the case....
 this->orb_->shutdown (0, ACE_TRY_ENV);
}

// ************************************************************
int main(int argc, char* argv[]) {

   // First initialize the ORB, that will remove some arguments...
   orb =
     CORBA::ORB_init (argc, argv,
                "myORB1" /* the ORB name, it can be anything! */);

          // Now call thread_main
          thread_main(argc, argv);

return 0;
}

// ************************************************************

int consumer_main(int &argc, char** argv)
```

```
{
 Consumer myconsumer;  // Define a consumer proxy object

std::cout << "Top of consumer_main\n";

 try {
std::cout << "Just inside try\n";

   // First initialize the ORB, that will remove some arguments...
   CORBA::ORB_var orb =
    CORBA::ORB_init (argc, argv,
                "myORB1" // the ORB name, it can be anything!
          );

std::cout << "after ORB_init\n";
    // Do *NOT* make a copy because we don't want the ORB to outlive
    // the Consumer object.
    myconsumer.orb_ = orb.in ();

    if (argc <= 1)
     {
                    std::cout << "too few parameters" << std::endl;
      return 1;
     }

    CORBA::Object_var object =
     orb->resolve_initial_references ("RootPOA");

    PortableServer::POA_var poa =
     PortableServer::POA::_narrow (object.in ());

    PortableServer::POAManager_var poa_manager =
     poa->the_POAManager ();
    poa_manager->activate ();


std::cout << "After POA activated\n";
    // Obtain the event channel, we could use a naming service, a
    // command line argument or resolve_initial_references(), but
    // this is simpler...
    object =
     orb->string_to_object (argv[1]);

std::cout << "after string to object\n";

    CosEventChannelAdmin::EventChannel_var event_channel =
     CosEventChannelAdmin::EventChannel::_narrow (object.in ());

    // The canonical protocol to connect to the EC
    CosEventChannelAdmin::ConsumerAdmin_var consumer_admin =
     event_channel->for_consumers ();

    CosEventChannelAdmin::ProxyPushSupplier_var supplier =
     consumer_admin->obtain_push_supplier ();

    CosEventComm::PushConsumer_var consumer =
```

```
      myconsumer._this ();

std::cout << "after servant activation\n";

    supplier->connect_push_consumer (consumer.in ());

std::cout << "Just before run\n";

    // Wait for events, using work_pending()/perform_work() may help
    // or using another thread, this example is too simple for that.
    orb->run ();

    // We don't do any cleanup, it is hard to do it after shutdown,
    // and would complicate the example; plus it is almost
    // impossible to do cleanup after ORB->run() because the POA is
    // in the holding state.  Applications should use
    // work_pending()/perform_work() to do more interesting stuff.
    // Check the supplier for the proper way to do cleanup.
  }
        catch (CORBA::Exception &) {
     std::cout << "General CORBA exception raised!" << std::endl;
          return 1;
        }

 return 0;

//  return consumer.run (argc, argv);
}
```

ietmmain.cpp

```
#include <windows.h>
#include <stdlib.h>
#include <iostream>
#include "ietm.h"

typedef struct {
        int argc;
        char *argv[100];
} thread_params;


extern int client (int &argc, char** argv);

// This thread runs the server waiting for callbacks
void ThreadOne(thread_params *args)
{


 consumer_main(args->argc, args->argv);
}


// This thread runs the client
void ThreadTwo(thread_params *args)
{
```

```
 Sleep(100);
 client (args->argc, args->argv);

}


// This file starts the threads
int thread_main(int argc, char* argv[])
{
 HANDLE ThreadHandles[2];
 DWORD ThreadOneID, ThreadTwoID;
//  DWORD j;
 int i;
 thread_params tmp;


  // Put the arguments in the struct so can pass to the server code
  tmp.argc=argc;

  i=0;
  while (i<argc) {
        tmp.argv[i]=argv[i];
        i++;
  }


  ThreadHandles[0] = CreateThread( 0,0,
          (LPTHREAD_START_ROUTINE) ThreadOne,
          &tmp, 0, &ThreadOneID);



        // Pause a good long while to give the server on the other side time to
   // start up before running the client on this side.
//   j = 0;
//        while (i<100) {

          Sleep(10);

//          j++;

//        }

 // Put the arguments in the struct so can pass to the client code
  tmp.argc=argc;

  i=0;
  while (i<argc) {
        tmp.argv[i]=argv[i];
        i++;
  }

  ThreadHandles[1] = CreateThread( 0,0,
          (LPTHREAD_START_ROUTINE) ThreadTwo,
          &tmp, 0, &ThreadTwoID);
```

```
        // Wait for all threads to finish execution
  WaitForMultipleObjects(2, ThreadHandles, TRUE, INFINITE);

//   WaitForMultipleObjects(1, ThreadHandles, TRUE, INFINITE);

        return 0;
}
```

quoter.idl

```
module echomodule
{

 exception Invalid_Message{};

 interface echo
 {
        string echostring(in string message)
                raises (Invalid_Message);
        short register_callback(in string myname);
 };
};
```

icemain.cpp

```
#include <windows.h>
#include <stdlib.h>
#include <iostream>

typedef struct {
        int argc;
        char *argv[100];
} thread_params;

extern void make_events();
extern int server (int &argc, char** argv);

int event;                      // Current event to send to remote
HANDLE semaphore0;
HANDLE semaphore1;

void release_semaphore0() {
 LONG semaCount;

  ReleaseSemaphore(semaphore0, 1, &semaCount);
}

void release_semaphore1() {
 LONG semaCount;

  ReleaseSemaphore(semaphore1, 1, &semaCount);
}

void wait_for_semaphore1() {
        WaitForSingleObject(semaphore1,INFINITE);
}
```

```
void wait_for_semaphore0() {
        WaitForSingleObject(semaphore0,INFINITE);
}


extern int supplier_main (int argc, char* argv[]);
extern void make_events();

void ThreadOne(thread_params *args)
{
 int i;

        wait_for_semaphore0();  // Block until at least one callback is registered
                                                                // before initializing the event service
        i=supplier_main(args->argc, args->argv);

}


// This thread makes arbitrary events
void ThreadTwo(thread_params *args)
{

        make_events();

} // end ThreadTwo


// This thread is a basic CORBA server
void ThreadThree(thread_params * args)
{
 int i;

 i=server(args->argc, args->argv);
}


// This file starts the threads
int thread_main(int argc, char* argv[])
{
 HANDLE ThreadHandles[3];
 DWORD ThreadOneID, ThreadTwoID, ThreadThreeID;
//  DWORD j;
 int i;
 thread_params tmp;

  // Put the arguments in the struct so can pass to the server code
  tmp.argc=argc;

  i=0;
  while (i<argc) {
        tmp.argv[i]=argv[i];
        i++;
  }

  // Create some semaphores
```

```
        // Start the semaphores off at zero, in order to initially block execution
  semaphore0 = CreateSemaphore(0,0,1,0);
  semaphore1 = CreateSemaphore(0,0,1,0);

  ThreadHandles[0] = CreateThread( 0,0,
            (LPTHREAD_START_ROUTINE) ThreadOne,
            &tmp, 0, &ThreadOneID);



        // Pause a good long while to give the server on the other side time to
   // start up before running the client on this side.
 //  j = 0;
//          while (i<100) {

            Sleep(10);

//          j++;

//          }



 // Put the arguments in the struct so can pass to the client code
  tmp.argc=argc;

  i=0;
  while (i<argc) {
        tmp.argv[i]=argv[i];
        i++;
  }

  ThreadHandles[1] = CreateThread( 0,0,
            (LPTHREAD_START_ROUTINE) ThreadTwo,
            &tmp, 0, &ThreadTwoID);

  ThreadHandles[2] = CreateThread( 0,0,
            (LPTHREAD_START_ROUTINE) ThreadThree,
            &tmp, 0, &ThreadThreeID);

        // Wait for all threads to finish execution
  WaitForMultipleObjects(3, ThreadHandles, TRUE, INFINITE);

//  WaitForMultipleObjects(1, ThreadHandles, TRUE, INFINITE);

  CloseHandle(semaphore0);
  CloseHandle(semaphore1);

  return 0;
}
```

server.cpp

```
#include "Stock_i.h"
#include "d:\TAO_ORB\TAO_zip_version\ACE_wrappers\TAO\orbsvcs\orbsvcs\CosNamingC.h"
#include <iostream>
```

```
extern CORBA::ORB_var orb;

int server (int &argc, char** argv)
{
 try {

//   CORBA::ORB_var orb =
//     CORBA::ORB_init (argc, argv,
//                      "myORB0" /* the ORB name, it can be anything! */);
   CORBA::Object_var poa_object =
    orb->resolve_initial_references ("RootPOA");
   PortableServer::POA_var poa =
    PortableServer::POA::_narrow (poa_object.in ());
   PortableServer::POAManager_var poa_manager =
    poa->the_POAManager ();
   poa_manager->activate ();

   // Create the servant
  // Quoter_Stock_Factory_i stock_factory_i;
         echo_i servant;

   // Activate it to obtain the object reference
//   Quoter::Stock_Factory_var stock_factory =
//     stock_factory_i._this ();
         echomodule::echo_var myobject=servant._this();

   // Get the Naming Context reference
//   CORBA::Object_var naming_context_object =
//     orb->resolve_initial_references ("NameService");
         CORBA::Object_var naming_context_object;
         try {
                  naming_context_object=
              orb->string_to_object (argv[2]);
         } catch (const CORBA::SystemException &se) {
                  std::cout << "Cannot resolve NameService address" << std::endl;
                  throw 0;
         }

   CosNaming::NamingContext_var naming_context =
    CosNaming::NamingContext::_narrow (naming_context_object.in ());

   // Create and initialize the name.
   CosNaming::Name name (1);
   name.length (1);
   name[0].id = CORBA::string_dup ("echo");

   // Bind the object
   naming_context->bind (name, myobject.in ());

   orb->run ();

   // Destroy the POA, waiting until the destruction terminates
   poa->destroy (1, 1);
   orb->destroy ();
 }
 catch (CORBA::Exception &) {
```

```
    std::cerr << "CORBA exception raised!" << std::endl;
  }
  return 0;
}
```

stock_i.cpp

```
#include "Stock_i.h"
#include <iostream>
//#include "ietm1.h"

extern void release_semaphore0();

echo_i::echo_i ()

{
}

char *
echo_i::echostring(const char * message) throw (CORBA::SystemException)
{
        std::cout << "***** Message received is " << message << std::endl;

        char * junkie=CORBA::string_dup(message);
        return junkie;
}

CORBA::Short echo_i::register_callback(const char * myname) {

        release_semaphore0();      // Attach to the event service if not already done

        std::cout << "***** The server has been notified that client " << myname;
        std::cout << "is now registered with the event service\n";

        return 0;
}
```

supplier.cpp

```
#include "Supplier.h"
//#include "orbsvcs/CosEventChannelAdminS.h"
#include
"d:\TAO_ORB\TAO_zip_version\ACE_wrappers\TAO\orbsvcs\orbsvcs\CosEventChannelAdminS.h"
#include <iostream>
using namespace std;

extern void release_semaphore0();
extern void release_semaphore1();
extern void wait_for_semaphore1();
extern int thread_main(int argc, char* argv[]);

CORBA::ORB_var orb;
CosEventChannelAdmin::ProxyPushConsumer_var consumer;

unsigned continue_looping;

CORBA::Any event;
```

```
void make_events() {
int current_event;
char response;

   continue_looping=1;  //Set up to loop while waiting for events

        while (continue_looping) {

                std::cout << "Do you want to send an event?\n";
                std::cout << "Y or y to send event, S or s to terminate\n";
                std::cout << "the event channel\n";
                std::cin >> response;
                if ((response=='y')|| (response=='Y')) {

//                      release_semaphore0();  // Go ahead and initialize the event service
                                                                        // for the first event

                        std::cout << "Enter integer event number to send\n";
                        std::cin >> current_event;

                        // Push the event
                        event <<= CORBA::ULong (current_event);


                        try {
                                consumer->push (event);
                        } catch (const CORBA::SystemException &se) {
                                std::cout << "the push didn't work" << std::endl;
                        }

                }
                else if ((response=='s')|| (response=='S')) {
                        continue_looping=0;  // Quit looping through sending events
                        // Fall through to destroy event channel and POA
                        release_semaphore1();
                }

        } // end while

}

int main (int argc, char *argv[]) {
int i;
        // ORB initialization boiler plate...
   orb =
      CORBA::ORB_init (argc, argv, "");

        // Now call thread_main
        i=thread_main(argc,argv);

        return i;
}

// *********************************************************************

Supplier::Supplier (void)
```

```
{
}

void
Supplier::disconnect_push_supplier (CORBA::Environment &)
{
}

// ************************************************************

int
supplier_main (int argc, char* argv[])
{
 Supplier mysupplier;  // Define the supplier proxy object

        try
  {

   if (argc <= 1)
    {
                  std::cout << "Too few parameters" << std::endl;
     return 1;
    }

   CORBA::Object_var object =
    orb->resolve_initial_references ("RootPOA");

   PortableServer::POA_var poa =
    PortableServer::POA::_narrow (object.in ());

   PortableServer::POAManager_var poa_manager =
    poa->the_POAManager ();

   poa_manager->activate ();

   // Obtain the event channel, we could use a naming service, a
   // command line argument or resolve_initial_references(), but
   // this is simpler...
   object =
    orb->string_to_object (argv[1]);


   CosEventChannelAdmin::EventChannel_var event_channel =
    CosEventChannelAdmin::EventChannel::_narrow (object.in ());


   // The canonical protocol to connect to the EC
   CosEventChannelAdmin::SupplierAdmin_var supplier_admin =
    event_channel->for_suppliers ();

        consumer =
    supplier_admin->obtain_push_consumer ();


   CosEventComm::PushSupplier_var supplier =
    mysupplier._this ();
```

```
    consumer->connect_push_supplier (supplier.in ());



                    // Note that it's also possible to push events here.


//std::cout << "Waiting for semaphore1\n";
            wait_for_semaphore1();


    // Disconnect from the EC
            try {
    consumer->disconnect_push_consumer ();
            } catch (const CORBA::SystemException &se) {
                    std::cout << "disconnect didn't work" << std::endl;
            }

    // Destroy the EC....
    event_channel->destroy ();


    // Deactivate the servant object...
//    PortableServer::ObjectId_var id =
//      poa->servant_to_id (this);

//    poa->deactivate_object (id.in ());


    // Destroy the POA
    poa->destroy (1, 0);

  }
        catch(CORBA::Exception &) {
                std::cout << "CORBA exception raised!" << std::endl;
                return 1;
        }

 return 0;

 // return supplier.run (argc, argv);
}
```

stock_i.h

```
#include "QuoterS.h"
#include <string>

class echo_i : public POA_echomodule::echo {
public:
 echo_i ();

 char *echostring (const char * message)
            throw (echomodule::Invalid_Message);
```

```
 CORBA::Short register_callback(const char * myname);

private:
};
```

supplier.h

```
#ifndef SUPPLIER_H
#define SUPPLIER_H

//#include "orbsvcs/CosEventCommS.h"
#include "d:\TAO_ORB\TAO_zip_version\ACE_wrappers\TAO\orbsvcs\orbsvcs\CosEventCommS.h"

#if !defined (ACE_LACKS_PRAGMA_ONCE)
# pragma once
#endif /* ACE_LACKS_PRAGMA_ONCE */

class Supplier : public POA_CosEventComm::PushSupplier
{
 // = TITLE
 //   Simple supplier object
 //
 // = DESCRIPTION
 //   This class is a supplier of events.
 //
public:
 Supplier (void);
 // Constructor

 // = The CosEventComm::PushSupplier methods

 virtual void disconnect_push_supplier (CORBA::Environment &);
 // The skeleton methods.

private:
};

#endif /* SUPPLIER_H */
```

Appendix D
CORBA Notification Service Demonstration

client.cpp

```
#include "QuoterC.h"
#include "TAO\orbsvcs\orbsvcs\CosNamingC.h"
#include "TAO\orbsvcs\orbsvcs\CosEventCommS.h"
#include "TAO\orbsvcs\orbsvcs\CosEventChannelAdminS.h"
#include <iostream>
#include <string>

#include "ClientFilterSetUp.h"
#include "Filter.h"

long AirQuality=0;
long Temperature=0;
long WaterQuality=0;


// global variables
CORBA::ORB_var orb;


typedef struct {
        int argc;
        char *argv[100];
} thread_params;



class SensorPackage {
public:
        SensorPackage();
        SensorPackage(CosNaming::NamingContext_ptr naming_context, std::string location);

        void process(const char *p);
private:
   std::string location;
   Sensors::AirQuality_var AQ;
        Sensors::WaterQuality_var WQ;
        Sensors::Temperature_var TEMP;
        CosEventChannelAdmin::EventChannel_var event_channel;
        CosEventComm::PushConsumer_var consumer;
};

SensorPackage::SensorPackage()
{
        std::cout << "SensorPackage does not have NULL constructor" << std::endl;
        assert(0);
}

SensorPackage::SensorPackage(CosNaming::NamingContext_ptr naming_context, std::string location)
{
        CosNaming::Name name (1);
        std::string fullname;
```

```
        this->location = location;
 try {
// START AirQuality

        fullname = location + "-AirQuality";
   name.length (1);
   name[0].id = CORBA::string_dup (fullname.c_str());

        CORBA::Object_var AQobj = naming_context->resolve (name);
        std::cout << "Resolved " << fullname.c_str() << std::endl;

   // Now downcast the object reference to the appropriate type
        try {
                AQ = Sensors::AirQuality::_narrow(AQobj);
        } catch (const CORBA::SystemException &se) {
                std::cerr << "Cannot narrow AQ reference" << std::endl;
                throw 0;
        }

        if ( CORBA::is_nil(AQ) ) {
                std::cout << "AQ is nil" << std::endl;
                assert(0);
        }
// END AirQuality

// START WaterQuality

   fullname = location + "-WaterQuality";
   name.length (1);
   name[0].id = CORBA::string_dup (fullname.c_str());

   CORBA::Object_var WQobj = naming_context->resolve (name);
   std::cout << "Resolved " << fullname.c_str() << std::endl;

   // Now downcast the object reference to the appropriate type
        try {
                WQ = Sensors::WaterQuality::_narrow(WQobj);
        } catch (const CORBA::SystemException &se) {
                std::cerr << "Cannot narrow WQ reference" << std::endl;
                throw 0;
        }

        if ( CORBA::is_nil(WQ) ) {
                std::cout << "WQ is nil" << std::endl;
                assert(0);
        }
// END WaterQuality

// START Temperature

   fullname = location + "-Temperature";
   name.length (1);
   name[0].id = CORBA::string_dup (fullname.c_str());

   CORBA::Object_var TEMPobj = naming_context->resolve (name);
```

```
        std::cout << "Resolved " << fullname.c_str() << std::endl;

  // Now downcast the object reference to the appropriate type
        try {
                TEMP = Sensors::Temperature::_narrow(TEMPobj);
        } catch (const CORBA::SystemException &se) {
                std::cerr << "Cannot narrow TEMP reference" << std::endl;
                throw 0;
        }

        if ( CORBA::is_nil(TEMP) ) {
                std::cout << "TEMP is nil" << std::endl;
                assert(0);
        }
// END Temperature


 }
 catch (const CORBA::Exception &) {
  std::cerr << "CORBA exception raised in SensorPackage instantiating "
                << location << std::endl;
 }

}

void SensorPackage::process(const char *p)
{
 CORBA::Char scale;
 Sensors::Temperature_t temperature;

 std::cout << "*** SensorPackage::process(" << p << ") at Lake " << location << std::endl;

 for ( ; *p ; p++ ) {
        switch (*p) {
/*    case 'a':
          Sensors::AirQuality::Values_t_slice *air;
                air = AQ->Get();
                std::cout << "Lake " << location << " AirQuality: (" <<
                  air[0] << ", " << air[1] << ", " <<
                  air[2] << ", " << air[3] << ")"  << std::endl;
                break;
*/
          case 'c':
                std::cout << "Lake " << location << " Configure: to Celcius" << std::endl;
                TEMP->Configure(Sensors::Celcius);
                break;
          case 'f':
                std::cout << "Lake " << location << " Configure: to Fahrenheit" << std::endl;
                TEMP->Configure(Sensors::Fahrenheit);
                break;
/*        case 't':
                temperature = TEMP->Get(scale);
                std::cout << "Lake " << location << " Temperature: " <<  temperature << scale <<
std::endl;
                break;
*/
```

```
/*          case 'w':
                    {
                     Sensors::Quality_t wq_north = WQ->Get("North");
                     Sensors::Quality_t wq_south = WQ->Get("South");
                     Sensors::Quality_t wq_east  = WQ->Get("East");
                     Sensors::Quality_t wq_west  = WQ->Get("West");
                     std::cout << "Lake " << location << " WaterQuality(NSEW): (" <<
                       wq_north << ", " << wq_south << ", " <<
                       wq_east  << ", " << wq_west  << ")"  << std::endl;
                    }
                    break;
*/
           default:
                    std::cout << "Unknown sensor code " << *p << std::endl;
                    exit(0);
                    break;
          }
          SleepEx(1000, 0 );
  }
}

#if 1
void RunORB()
{
                    CosEventChannelAdmin::EventChannel_var event_channel;
          CosEventComm::PushConsumer_var consumer;

          CosNaming::Name name (1);
CosNaming::NamingContext_ptr naming_context;

           char * argv[] = {"cmd", "file://iorfile", 0 };
          int argc = 1;

  orb = CORBA::ORB_init (argc, argv,
                "" /* the ORB name, it can be anything! */);
          CORBA::Object_var naming_context_object = orb->string_to_object(argv[1]);

          // Because we pass the naming context in to a subroutine, it is
          // necessary to declare it as a _ptr rather than an _var.  This
          // appears to impact getting it initialized properly.
   naming_context =
    CosNaming::NamingContext::_narrow (naming_context_object.in ());

          // activate a Root POA
          CORBA::Object_var poa_object = orb->resolve_initial_references("RootPOA");
          PortableServer::POA_var poa = PortableServer::POA::_narrow(poa_object.in());
          PortableServer::POAManager_var poa_manager = poa->the_POAManager();
          poa_manager->activate();

     std::cout << "Starting the ORB Thread" << std::endl;
   orb->run();
}
#endif

int my_argc;
char **my_argv;
```

```
CosNaming::NamingContext_ptr naming_context;
void RunCmds()
{
        int argc = my_argc;
        char ** argv= my_argv;
        int arg;
  SensorPackage LakeGuntersville(naming_context, "Guntersville");
        SensorPackage LakeLanier(naming_context, "Lanier");

        for (arg=2 ; arg<argc ; arg++ ) {
                std::cout << "Processing " << argv[arg] << std::endl;
                if ( argv[arg][0] != '-') {
                  std::cout << "first char of arg is not a - : " << argv[arg] << std::endl;
                  exit(0);
                }

                switch ( argv[arg][1] ) {
                  case 'g':         // sensors at Lake Guntersville
                    LakeGuntersville.process(&argv[arg][2]);
                        break;
                  case 'l':         // sensors at Lake Lanier
       LakeLanier.process(&argv[arg][2]);
                        break;
                  default:
                        std::cout << "Unknown location code " << argv[arg][1] << std::endl;
                        exit(0);
                        break;
                }
        }

}

// This thread runs the ORB.
void ThreadOne()
{
        orb->run();
}

void ThreadTwo(thread_params *args)
{
 FilterClient filterclient;

 try {

        filterclient.init (args->argc, args->argv); //Init the Client
std::cout << "In ThreadTwo, just below filterclient.init" << std::endl;

#if SUPPLIER
   filterclient.run ();                         // Send some events locally
#endif

 }
 catch (CORBA::UserException &ue)
  {
          std::cout << "User exception" << std::endl;
```

```
    throw 0;
   }
 catch (CORBA::SystemException  &se)
   {
          std::cout << "System exception" << std::endl;
    throw 0;
   }
std::cout << "In ThreadTwo, just before while 1" << std::endl;

while (1) {
}

}

void ThreadThree()
{

}

// real client main
int main (int argc, char* argv[])
{

std::cout << "Top of main" << std::endl;
 try  {
   // First initialize the ORB, that will remove some arguments...
   orb = CORBA::ORB_init (argc, argv,
                "" /* the ORB name, it can be anything! */);


          CORBA::Object_var naming_context_object = orb->string_to_object(argv[1]);

          // Because we pass the naming context in to a subroutine, it is
          // necessary to declare it as a _ptr rather than an _var.  This
          // appears to impact getting it initialized properly.
   naming_context =
    CosNaming::NamingContext::_narrow (naming_context_object.in ());

          // activate a Root POA
          CORBA::Object_var poa_object = orb->resolve_initial_references("RootPOA");
          PortableServer::POA_var poa = PortableServer::POA::_narrow(poa_object.in());
          PortableServer::POAManager_var poa_manager = poa->the_POAManager();
          poa_manager->activate();

          // instantiate Sensor Packages at the lakes
          // SensorPackage LakeGuntersville(naming_context, "Guntersville");
          // SensorPackage LakeLanier(naming_context, "Lanier");

   // start the ORB running in another thread
          HANDLE ThreadHandles[4];
   DWORD RunCmdsThreadID, ThreadOneID, ThreadTwoID, ThreadThreeID;
          thread_params tmp;
          int i;

          my_argc = argc;
          my_argv = argv;
```

```
  ThreadHandles[0] = CreateThread( 0,0,
            (LPTHREAD_START_ROUTINE) RunCmds,
            0, 0, &RunCmdsThreadID);

  ThreadHandles[1] = CreateThread( 0,0,
            (LPTHREAD_START_ROUTINE) ThreadOne,
            0, 0, &ThreadOneID);

 // Put the arguments in the struct so can pass to the client code
  tmp.argc=argc;

  i=0;
  while (i<argc) {
          tmp.argv[i]=argv[i];
          i++;
  }

  ThreadHandles[2] = CreateThread( 0,0,
            (LPTHREAD_START_ROUTINE) ThreadTwo,
            &tmp, 0, &ThreadTwoID);

  ThreadHandles[3] = CreateThread( 0,0,
            (LPTHREAD_START_ROUTINE) ThreadThree,
            0, 0, &ThreadThreeID);


std::cout << "Very Bottom! just before threads finish" << std::endl;
          // Wait for all threads to finish execution
  WaitForMultipleObjects(4, ThreadHandles, TRUE, INFINITE);



          // Now parse the arguments and fetch sensor data

#if 0
          int arg;

          for (arg=2 ; arg<argc ; arg++ ) {
                    std::cout << "Processing " << argv[arg] << std::endl;
                    if ( argv[arg][0] != '-') {
                      std::cout << "first char of arg is not a - : " << argv[arg] << std::endl;
                      exit(0);
                    }

                    switch ( argv[arg][1] ) {
                      case 'g':         // sensors at Lake Guntersville
                        LakeGuntersville.process(&argv[arg][2]);
                              break;
                      case 'l':         // sensors at Lake Lanier
        LakeLanier.process(&argv[arg][2]);
                              break;
                      default:
                              std::cout << "Unknown location code " << argv[arg][1] << std::endl;
                              exit(0);
                              break;
```

```
                }
        }
#endif
        // TerminateThread( ThreadHandle, 0 );

  // Finally destroy the ORB -- blows up some TAO versions
  // orb->destroy ();
 }
 catch (const CORBA::Exception &) {
  std::cerr << "CORBA exception raised!" << std::endl;
 }

 return 0;
}
```

consumerfilter.cpp

```
#include "orbsvcs/orbsvcs/Notify/Notify_StructuredPushConsumer.h"
#include "orbsvcs/orbsvcs/Notify/Notify_StructuredPushSupplier.h"

#include "ClientFilterSetUp.h"
#include "Filter.h"

#include <iostream>
#include <stdlib.h>

#define NOTIFY_FACTORY_NAME "NotifyEventChannelFactory"
#define NAMING_SERVICE_NAME "NameService"
#define CA_FILTER "Temperature < 70"
#define SA_FILTER "Temperature > 10"
#define TCL_GRAMMER "TCL"
#define MAX_STRING_SIZE 256
#define NOTIFY_CHANNEL "myNotifyChannel"

extern long AirQuality;
extern long Temperature;
extern long WaterQuality;

class PushConsumer : public TAO_Notify_StructuredPushConsumer
{
private:
 char myname_[MAX_STRING_SIZE];


public:
 PushConsumer (const char* myname)
 {
        strcpy(myname_,myname);

 }

 virtual void push_structured_event (
    const CosNotification::StructuredEvent & notification,
    CORBA::Environment &/*ACE_TRY_ENV*/
    )
  ACE_THROW_SPEC ((
            CORBA::SystemException,
```

```
            CosEventComm::Disconnected
            ))
 {
  CORBA::Long val;
        CORBA::Long value;

  notification.remainder_of_body >>= val;

        std::cout <<  myname_ << " received event " << val << ":" << std::endl;
        for (int i=0; i<notification.filterable_data.length(); i++) {

                std::cout << "  " << (notification.filterable_data[i].name).in() << std::endl;

                if (notification.filterable_data[i].value >>= value)
                        std::cout << "  " << value << std::endl;

        }
 }
};

class PushSupplier : public TAO_Notify_StructuredPushSupplier
{
private:
char myname_[MAX_STRING_SIZE];

public:
 PushSupplier (const char* myname)
 {
        strcpy(myname_,myname);
 }

virtual void send_event (const CosNotification::StructuredEvent& event,
                CORBA::Environment &ACE_TRY_ENV)
 {
std::cout << "Top of PushSupplier::send_event" << std::endl;
//   ACE_DEBUG ((LM_DEBUG,
//             "%s is sending an event \n", myname_.fast_rep ()));
          std::cout << myname_ << " is sending an event" << std::endl;

   // add you own constraints here.
   TAO_Notify_StructuredPushSupplier::send_event (event,
                                  ACE_TRY_ENV);

 }
};

FilterClient::FilterClient (void)
{
 // No-Op.
 ifgop_ = CosNotifyChannelAdmin::OR_OP;
}

FilterClient::~FilterClient ()
{
 // No-Op.
}
```

```
void
FilterClient::init (int argc, char *argv [])
{

std::cout << "In client FilterClient::init, top" << std::endl;

 init_ORB (argc, argv);
std::cout << "In client FilterClient::init, after init_orb" << std::endl;

 resolve_naming_service ();
std::cout << "In client FilterClient::init, after resolve_naming_service" << std::endl;

#if SUPPLIER
// Set up the Notify factory, and create the event channel
 resolve_Notify_factory ();
std::cout << "In client FilterClient::init, after resolve_Notify_factory" << std::endl;

 create_EC ();
std::cout << "In client FilterClient::init, after create_EC" << std::endl;
#endif

#if CONSUMER
 resolve_Notify_Channel();
 std::cout << "In client FilterClient::init, after resolve_Notify_Channel" << std::endl;
#endif



#if SUPPLIER
 create_supplieradmin ();
#endif

#if CONSUMER
 create_consumeradmin ();
#endif
std::cout << "In client FilterClient::init, after create_consumeradmin" << std::endl;

#if CONSUMER
 create_consumers ();
#endif
std::cout << "In client FilterClient::init, after create_consumers" << std::endl;

#if SUPPLIER
 create_suppliers ();
#endif

std::cout << "In client FilterClient::init, very bottom" << std::endl;

}

void
FilterClient::run ()
{
 send_events ();
}
```

```
void
FilterClient::init_ORB (int argc,
                char *argv [])
{
 this->orb_ = CORBA::ORB_init (argc,
                        argv,
                        "");

 CORBA::Object_ptr poa_object =
  this->orb_->resolve_initial_references("RootPOA");


 if (CORBA::is_nil (poa_object))
  {
//    ACE_ERROR ((LM_ERROR,
//                " (%P|%t) Unable to initialize the POA.\n"));
    return;
  }
 this->root_poa_ =
  PortableServer::POA::_narrow (poa_object);


 PortableServer::POAManager_var poa_manager =
  root_poa_->the_POAManager ();


 poa_manager->activate ();

}

void
FilterClient::resolve_naming_service ()
{
 CORBA::Object_var naming_obj =
  this->orb_->resolve_initial_references (NAMING_SERVICE_NAME);


 // Need to check return value for errors.
 if (CORBA::is_nil (naming_obj.in ()))
  //ACE_THROW (CORBA::UNKNOWN ());
        throw 0;

 this->naming_context_ =
  CosNaming::NamingContext::_narrow (naming_obj.in ());

}

void
FilterClient::resolve_Notify_factory ()
{
 CosNaming::Name name (1);
 name.length (1);
 name[0].id = CORBA::string_dup (NOTIFY_FACTORY_NAME);

 CORBA::Object_var obj =
```

```
  this->naming_context_->resolve (name);

 this->notify_factory_ =
  CosNotifyChannelAdmin::EventChannelFactory::_narrow (obj.in ());

}

void
FilterClient::create_EC ()
{
 CosNotifyChannelAdmin::ChannelID id;
 CosNaming::Name name (1);

 ec_ = notify_factory_->create_channel (initial_qos_,
                                        initial_admin_,
                                        id);
 try {
          this->naming_context_->bind (name, ec_.in ());
 }
 catch (const CosNaming::NamingContext::AlreadyBound &) {
          std::cout << "Earlier server already bound the NOTIFY_CHANNEL" << std::endl;
          std::cout << "That's okay, just proceed using its value" << std::endl;

          // Have to resolve the notify channel, instead of using the new
          // event channel just created
          resolve_Notify_Channel();

 }

// ACE_ASSERT (!CORBA::is_nil (ec_.in ()));
 if (CORBA::is_nil (ec_.in())) {
          std::cout << "event channel is nil\n";
          throw 0;
 }
}

void FilterClient:: resolve_Notify_Channel()
{
        CosNaming::Name name;
        name.length(1);
        name[0].id = CORBA::string_dup(NOTIFY_CHANNEL);

        CORBA::Object_var notify_channel_obj = this->naming_context_->resolve(name);

        ec_ = CosNotifyChannelAdmin::EventChannel::_narrow(notify_channel_obj.in());

        if (CORBA::is_nil(ec_.in())) {
                std::cout << "Unable to find the notification channel" << std::endl;
        }

}


void
FilterClient::create_supplieradmin ()
{
```

```
  CosNotifyChannelAdmin::AdminID adminid = 0;

  supplier_admin_ =
    ec_->new_for_suppliers (this->ifgop_, adminid);


// ACE_ASSERT (!CORBA::is_nil (supplier_admin_.in ()));
  if (CORBA::is_nil(supplier_admin_.in())) {
            std::cout << "supplier_admin is nil" << std::endl;
            throw 0;
  }

  CosNotifyFilter::FilterFactory_var ffact =
    ec_->default_filter_factory ();


  // setup a filter at the consumer admin
  CosNotifyFilter::Filter_var sa_filter =
    ffact->create_filter (TCL_GRAMMER);

// ACE_ASSERT (!CORBA::is_nil (sa_filter.in ()));
  if (CORBA::is_nil(sa_filter.in())) {
            std::cout << "sa_filter is nil" << std::endl;
            throw 0;
  }

  CosNotifyFilter::ConstraintExpSeq constraint_list (1);
  constraint_list.length (1);

  constraint_list[0].event_types.length (0);
  constraint_list[0].constraint_expr = CORBA::string_dup (SA_FILTER);

  sa_filter->add_constraints (constraint_list);

  supplier_admin_->add_filter (sa_filter.in ());

}

void
FilterClient:: create_consumeradmin ()
{
  CosNotifyChannelAdmin::AdminID adminid = 0;

  consumer_admin_ =
    ec_->new_for_consumers (this->ifgop_, adminid);


// ACE_ASSERT (!CORBA::is_nil (consumer_admin_.in ()));
  if (CORBA::is_nil(consumer_admin_.in())) {
            std::cout << "consumer admin is nil" << std::endl;
            throw 0;
  }

  CosNotifyFilter::FilterFactory_var ffact =
    ec_->default_filter_factory ();
```

```
 // setup a filter at the consumer admin
 CosNotifyFilter::Filter_var ca_filter =
  ffact->create_filter (TCL_GRAMMER);

//  ACE_ASSERT (!CORBA::is_nil (ca_filter.in ()));
 if (CORBA::is_nil(ca_filter.in())) {
          std::cout << "ca_filter is nil" << std::endl;
          throw 0;
 }

 /*  struct ConstraintExp {
          CosNotification::EventTypeSeq event_types;
          string constraint_expr;
          };
 */
 CosNotifyFilter::ConstraintExpSeq constraint_list (1);
 constraint_list.length (1);

 constraint_list[0].event_types.length (0);
 constraint_list[0].constraint_expr = CORBA::string_dup (CA_FILTER);

 ca_filter->add_constraints (constraint_list);


 consumer_admin_->add_filter (ca_filter.in ());

}

void
FilterClient::create_consumers ()
{

 // startup the first consumer.
//  ACE_NEW_THROW_EX (consumer_1,
//              PushConsumer ("consumer1"),
//              CORBA::NO_MEMORY ());
 do { try { consumer_1 = new PushConsumer ("consumer1") ; }
 catch (void *all) {
          std::cout << "Not enough memory" << std::endl;
 }
 } while (0);


ACE_TRY_NEW_ENV
 consumer_1->init (this->root_poa_.in (), ACE_TRY_ENV);

 consumer_1->connect (consumer_admin_.in (), ACE_TRY_ENV);
ACE_CATCH (CORBA::SystemException, se);
ACE_ENDTRY;


 // startup the second consumer.
//  ACE_NEW_THROW_EX (consumer_2,
//              PushConsumer ("consumer2"),
//              CORBA::NO_MEMORY ());
 do { try { consumer_2 = new PushConsumer ("consumer1") ; }
```

```
 catch (void *all) {
          std::cout << "Not enough memory" << std::endl;
 }
 } while (0);

ACE_TRY_NEW_ENV
 consumer_2->init (this->root_poa_.in (), ACE_TRY_ENV);

 consumer_2->connect (consumer_admin_.in (), ACE_TRY_ENV);
ACE_CATCH (CORBA::SystemException, se);
ACE_ENDTRY;
}

void
FilterClient::create_suppliers ()
{
 // startup the first supplier
//  ACE_NEW_THROW_EX (supplier_1,
//                PushSupplier ("supplier1"),
//                CORBA::NO_MEMORY ());
 do { try { supplier_1 = new PushSupplier("supplier1") ; }
 catch (void *all) {
          std::cout << "Not enough memory" << std::endl;
 }
 } while (0);


ACE_TRY_NEW_ENV
 supplier_1->init (this->root_poa_.in (), ACE_TRY_ENV);

 supplier_1->connect (supplier_admin_.in (), ACE_TRY_ENV);
ACE_CATCH (CORBA::SystemException, se);
ACE_ENDTRY;

 // startup the second supplier
//  ACE_NEW_THROW_EX (supplier_2,
//                PushSupplier ("supplier2"),
//                CORBA::NO_MEMORY ());
 do { try { supplier_2 = new PushSupplier("supplier2") ; }
 catch (void *all) {
          std::cout << "Not enough memory" << std::endl;
 }
 } while (0);


ACE_TRY_NEW_ENV
 supplier_2->init (this->root_poa_.in (), ACE_TRY_ENV);

 supplier_2->connect (supplier_admin_.in (), ACE_TRY_ENV);
ACE_CATCH (CORBA::SystemException, se);
ACE_ENDTRY;

}

void
FilterClient::send_events ()
```

```
{
 // operations:
 CosNotification::StructuredEvent event;



 // EventHeader

 // FixedEventHeader
 // EventType
 // string
 event.header.fixed_header.event_type.domain_name = CORBA::string_dup("*");
 // string
 event.header.fixed_header.event_type.type_name = CORBA::string_dup("*");
 // string
 event.header.fixed_header.event_name = CORBA::string_dup("myevent");

 // OptionalHeaderFields
 // PropertySeq
 // sequence<Property>: string name, any value
 event.header.variable_header.length (1); // put nothing here

 // FilterableEventBody
 // PropertySeq
 // sequence<Property>: string name, any value
 event.filterable_data.length (3);
 event.filterable_data[0].name = CORBA::string_dup("AirQuality");

 event.filterable_data[1].name = CORBA::string_dup("Temperature");

 event.filterable_data[2].name = CORBA::string_dup("WaterQuality");

 for (int i = 0; i < 30; i++)
  {
   event.filterable_data[0].value <<= (CORBA::Long)AirQuality;
         event.filterable_data[1].value <<= (CORBA::Long)Temperature;
         event.filterable_data[2].value <<= (CORBA::Long)WaterQuality;


         // any -- let remainder of the body send the event #
   event.remainder_of_body <<= (CORBA::Long)i;


ACE_TRY_NEW_ENV
   supplier_1->send_event (event, ACE_TRY_ENV);

   supplier_2->send_event (event, ACE_TRY_ENV);
ACE_CATCH (CORBA::SystemException, se);
ACE_ENDTRY;
  }

}
```

clientfiltersetup.h

```
#define CONSUMER 1
//#define SUPPLIER 1
```

filter.h

```
#ifndef NOTIFY_FILTER_CLIENT_H
#define NOTIFY_FILTER_CLIENT_H

#include "orbsvcs/orbsvcs/CosNotifyChannelAdminC.h"
#include "orbsvcs/orbsvcs/CosNotifyCommC.h"
#include "orbsvcs/orbsvcs/CosNamingC.h"

class TAO_Notify_StructuredPushConsumer;
class TAO_Notify_StructuredPushSupplier;

class FilterClient
{
 // = TITLE
 //   Filter Client
 // = DESCRIPTION
 //   Client example that shows how to do Structured Event filtering
 //   in the Notification Service.

public:
 // = Initialization and Termination
 FilterClient (void);
 ~FilterClient ();

 void init (int argc, char *argv []);
 // Init the Client.

 void run ();
 // Run the demo.

protected:
 void init_ORB (int argc, char *argv []);
 // Initializes the ORB.

 void resolve_naming_service ();
 // Try to get hold of a running naming service.

 void resolve_Notify_factory ();
 // Try to resolve the Notify factory from the Naming service.

 void create_EC ();
 // Create an EC.

 void resolve_Notify_Channel();

 void create_supplieradmin();
 // Create the Supplier Admin.

 void create_consumeradmin ();
 // Create the Consumer Admin.

 void create_consumers ();
 // Create and initialize the consumers.

 void create_suppliers ();
```

```
// create and initialize the suppliers.

void send_events ();
// send the events.

// = Data Members
PortableServer::POA_var root_poa_;
// Reference to the root poa.

CORBA::ORB_var orb_;
// The ORB that we use.

CosNaming::NamingContext_var naming_context_;
// Handle to the name service.

CosNotifyChannelAdmin::EventChannelFactory_var notify_factory_;
// Channel factory.

CosNotifyChannelAdmin::EventChannel_var ec_;
// The one channel that we create using the factory.

CosNotifyChannelAdmin::InterFilterGroupOperator ifgop_;
// The group operator between admin-proxy's.

CosNotification::QoSProperties initial_qos_;
// Initial qos specified to the factory when creating the EC.

CosNotification::AdminProperties initial_admin_;
// Initial admin props specified to the factory when creating the EC.

CosNotifyChannelAdmin::ConsumerAdmin_var consumer_admin_;
// The consumer admin used by consumers.

CosNotifyChannelAdmin::SupplierAdmin_var supplier_admin_;
// The supplier admin used by suppliers.

TAO_Notify_StructuredPushConsumer* consumer_1;
// Consumer #1

TAO_Notify_StructuredPushConsumer* consumer_2;
// Consumer #2

TAO_Notify_StructuredPushSupplier* supplier_1;
// Supplier #1

TAO_Notify_StructuredPushSupplier* supplier_2;
// Supplier #2
};
#endif /* NOTIFY_FILTER_CLIENT_H */
```

quoter.idl

```
module Sensors
{


// 0-10 where 10 is highest quality and 0 is lowest
```

```
typedef short Quality_t;

// temperature scale is either Fahrenheit or Celcius
enum TemperatureScale_t { Fahrenheit, Celcius };

typedef float Temperature_t;

 interface AirQuality
 {
        // read at 0, 6, 20, 100 feet
        typedef Quality_t Values_t[4];

        Values_t Get();
 };

 interface WaterQuality
 {
  // valid quadrants are East, West, North, South
        exception Invalid_Quadrant{ };

        Quality_t Get(in string quadrant)
                raises (Invalid_Quadrant);
 };


 interface Temperature
 {
  struct TemperatureDropEvent {
          string      location;
          Temperature_t old;
          Temperature_t new;
         };

        void Configure(in TemperatureScale_t scale);

        // scale = 'F' or 'C'
        Temperature_t Get(out char scale);
 };

};
```

server.cpp

```
#include "Stock_i.h"
#include "TAO\orbsvcs\orbsvcs\CosNamingC.h"

#include <iostream>
#include <string>

#include "ServerFilterSetup.h"
#include "Filter.h"


long AirQuality=0;
long Temperature=0;
long WaterQuality=0;
```

```
typedef struct {
        int argc;
        char *argv[100];
} thread_params;


std::string location;
CosNaming::NamingContext_var naming_context;
CORBA::ORB_var orb;

// This thread runs the ORB, and waits for client requests.
void ThreadOne()
{

// Now Start the ORB and let Servants reply to requests
  orb->run ();

  orb->destroy ();
}

void ThreadTwo(thread_params *args)
{
 FilterClient filterclient;

 try {

        filterclient.init (args->argc, args->argv); //Init the Client

        Sleep(1000);
  filterclient.run ();                    // Send some events

 }
 catch (CORBA::UserException &ue)
  {

   throw 0;
  }
 catch (CORBA::SystemException  &se)
  {

   throw 0;
  }


}

// Simulate varying values of AirQuality, Temperature, and WaterQuality
void ThreadThree()
{

        unsigned i;

        AirQuality=0;
        Temperature=0;
        WaterQuality=0;
```

```
        i=0;
        while (1) {

                AirQuality=i;
                Temperature=i;
                WaterQuality=i;
                i++;
                if (i>90)
                        i=0;
        }

}

int main (int argc, char* argv[])
{
 std::string fullname;
 Sensors::Quality_t AirQuality;
 Sensors::Quality_t WaterQuality;
 Sensors::Temperature_t Temperature;
 CosNaming::Name name (1);

 HANDLE ThreadHandles[3];
 DWORD ThreadOneID, ThreadTwoID, ThreadThreeID;


 try {
  // First initialize the ORB, that will remove some arguments...
  orb =
   CORBA::ORB_init (argc, argv,
                "" /* the ORB name, it can be anything! */);

  // Get the Naming Context reference
  CORBA::Object_var naming_context_object =
   orb->resolve_initial_references ("NameService");
  naming_context =
   CosNaming::NamingContext::_narrow (naming_context_object.in ());

        // activate a Root POA
        CORBA::Object_var poa_object = orb->resolve_initial_references("RootPOA");
        PortableServer::POA_var poa = PortableServer::POA::_narrow(poa_object.in());
        PortableServer::POAManager_var poa_manager = poa->the_POAManager();
        poa_manager->activate();

std::cout << "just activated POA manager" << std::endl;

        // parse the arguments after the ORB has eaten its arguments
  assert( argc != 4 );
        location = (std::string) argv[1];
        AirQuality = (Sensors::Quality_t) atoi(argv[2]);
  WaterQuality = (Sensors::Quality_t) atoi(argv[3]);
        Temperature = (Sensors::Temperature_t) atoi(argv[4]);

std::cout << "just parsed arguments" << std::endl;

// START AirQuality Sensor
  // Create the servant
```

```
        AirQuality_i AirQualitySensor( AirQuality );

    // Activate it to obtain the object reference
        Sensors::AirQuality_var AQobject = AirQualitySensor._this();

    // Create and initialize the name.
        fullname = location + "-AirQuality";
    name.length (1);
    name[0].id = CORBA::string_dup (fullname.c_str());

        std::cout << "Registering: " << fullname << std::endl;
    // Bind the object
    naming_context->bind (name, AQobject.in ());
// END AirQuality Sensor

std::cout << "just created AirQuality Sensor" << std::endl;

// START WaterQuality Sensor
    // Create the servant
        WaterQuality_i WaterQualitySensor( WaterQuality );

    // Activate it to obtain the object reference
        Sensors::WaterQuality_var WQobject = WaterQualitySensor._this();

    // Create and initialize the name.
        fullname = location + "-WaterQuality";
    name.length (1);
    name[0].id = CORBA::string_dup (fullname.c_str());

        std::cout << "Registering: " << fullname << std::endl;
    // Bind the object
    naming_context->bind (name, WQobject.in ());
// END WaterQuality Sensor

std::cout << "just created WaterQuality Sensor" << std::endl;

// START Temperature Sensor
    // Create the servant
        Temperature_i TemperatureSensor( Temperature );
std::cout << "temp1" << std::endl;
    // Activate it to obtain the object reference
        Sensors::Temperature_var TEMPobject = TemperatureSensor._this();
std::cout << "temp2" << std::endl;
    // Create and initialize the name.
        fullname = location + "-Temperature";
    name.length (1);
    name[0].id = CORBA::string_dup (fullname.c_str());
std::cout << "temp3" << std::endl;
        std::cout << "Registering: " << fullname << std::endl;
    // Bind the object
    naming_context->bind (name, TEMPobject.in ());
// END Temperature Sensor

std::cout << "just created TemperatureQuality Sensor" << std::endl;

    ThreadHandles[0] = CreateThread( 0,0,
```

```
            (LPTHREAD_START_ROUTINE) ThreadOne,
            0, 0, &ThreadOneID);

  int i;
  thread_params tmp;


  // Put the arguments in the struct so can pass to the server code
  tmp.argc=argc;

  i=0;
  while (i<argc) {
          tmp.argv[i]=argv[i];
          i++;
  }


  ThreadHandles[1] = CreateThread( 0,0,
            (LPTHREAD_START_ROUTINE) ThreadTwo,
            &tmp, 0, &ThreadTwoID);

  ThreadHandles[2] = CreateThread( 0,0,
            (LPTHREAD_START_ROUTINE) ThreadThree,
            0, 0, &ThreadThreeID);

          // Wait for all threads to finish execution
  WaitForMultipleObjects(3, ThreadHandles, TRUE, INFINITE);

 }
 catch (CORBA::Exception &) {
  std::cerr << "CORBA exception raised!" << std::endl;
 }
 return 0;
}
```

stock_i.cpp

```
#include "Stock_i.h"
#include <iostream>
#include <string>
#include "TAO\orbsvcs\orbsvcs\CosNamingC.h"  // for DynAnyFactory
#include "DynAnyC.h"

extern std::string location;  // where are we???
extern CosNaming::NamingContext_var naming_context;

// START IDL: Sensors::AirQuality
AirQuality_i::AirQuality_i()
{
          AirQuality_i(0);
}

AirQuality_i::AirQuality_i(Sensors::Quality_t quality)
{
          if (quality > 10 || quality < 0 )
            throw 0;
          current = quality;
```

```
}

Sensors::AirQuality::Values_t_slice * AirQuality_i::Get()
{
        int i;
        Sensors::AirQuality::Values_t_slice * return_slice;
        return_slice = Sensors::AirQuality::Values_t_alloc();

   std::cout << "AirQuality::Get ==> ";

        for (i=0; i<4; i++) {
                std::cout << current << " ";
                return_slice[i] = current;
                current++;
                if (current > 10)
                        current = 0;
        }
        std::cout << std::endl;

        return return_slice;
}
// END IDL: Sensors::AirQuality

// START IDL: Sensors::WaterQuality
WaterQuality_i::WaterQuality_i()
{
   WaterQuality_i(0);
}

WaterQuality_i::WaterQuality_i(Sensors::Quality_t quality)
{
        if (quality > 10 || quality < 0 )
          throw 0;
        current = quality;
}

Sensors::Quality_t
WaterQuality_i::Get(const char * quadrant)
throw (Sensors::WaterQuality::Invalid_Quadrant)
{
        Sensors::Quality_t return_value;

        if ( !strcmp(quadrant, "East" ) &&
            !strcmp(quadrant, "West" ) &&
            !strcmp(quadrant, "North" ) &&
                 !strcmp(quadrant, "South" ) ) {
                std::cout << "WaterQuality::Get -- Invalid quadrant: " << quadrant << std::endl;
                throw Sensors::WaterQuality::Invalid_Quadrant();
        }

        return_value = current;
        current++;
        if (current > 10)
          current = 0;
   std::cout << "WaterQuality::Get(" << quadrant << ") ==> " << return_value << std::endl;
```

```
        return return_value;
}
// END IDL: Sensors::WaterQuality

// START IDL: Sensors::Temperature
Temperature_i::Temperature_i()
{
        Temperature_i( (Sensors::Temperature_t) 90.0);
}

Temperature_i::Temperature_i(Sensors::Temperature_t temperature)
{
        current_temperature = temperature;
        current_scale = 'F';

}

// void Configure(in TemperatureScale_t scale);
void Temperature_i::Configure(Sensors::TemperatureScale_t scale)
{
        std::cout << "Temperature::Configure(" << scale << ")" << std::endl;
        if ( scale == Sensors::Fahrenheit )
          current_scale = 'F';
        else
          current_scale = 'C';
}

// scale = 'F' or 'C'
// Temperature_t Get(out char scale);
Sensors::Temperature_t Temperature_i::Get(CORBA::Char_out scale)
{
        Sensors::Temperature_t return_temperature;

        return_temperature = current_temperature;
   scale = current_scale;

        current_temperature += 10.0;
        if ( current_temperature > 100.0 ) {
          current_temperature -= 80.0;

          // now generate the sudden change event

          std::cout << "ALERT: Temperature was " << return_temperature <<
                  " is " << current_temperature << std::endl;


          Sensors::Temperature::TemperatureDropEvent te;
          // te.location = CORBA::string_dup (location.c_str());
          // te.old = return_temperature;
          // te._cxx_new = current_temperature;

          CORBA::Any any;
          //any <<= te;
           any <<= (CORBA::Short) 16;

#if 0
```

```
        CosNaming::Name name (1);
    name.length (1);
    name[0].id = CORBA::string_dup ("DynAnyFactory");
        // Resolve the binding to the dynanyfactory reference

        DynamicAny::DynAnyFactory_var daf;
        try {
         CORBA::Object_var daf_object = naming_context->resolve( name );
         if ( CORBA::is_nil(daf_object) ) {
                        std::cout << "Nil DynAnyFactory raw object" << std::endl;
                        throw 0;
                }
         daf =
                DynamicAny::DynAnyFactory::_narrow(daf_object.in());
         if ( CORBA::is_nil(daf) ) {
                        std::cout << "Nil DynAnyFactory" << std::endl;
                        throw 0;
                }
        } catch (CORBA::Exception &) {
      std::cerr << "CORBA exception raised for DynAnyFactory! " << std::endl;
        }
         DynamicAny::DynAny_var da =
                daf->create_dyn_any_from_type_code(TemperatureDropEvent_tc);
         DynamicAny::DynStruct_var ds =
                DyanamicAny::DynStruct::_narrow(da);
         DynamicAny::DynAny_var member;
         member = ds->current_component();
         member->insert_string("XXX");
         ds->next();
         member->insert_float( 1.0 );
         ds->next();
         member->insert_float( 2.0 );

         CORBA::Any_var any = ds->to_any();

#endif
         std::cout << "here we go" << std::endl;
         consumer->push(any);
        }

    if ( current_scale == 'C' )
         return_temperature = (return_temperature - (Sensors::Temperature_t)32) * 5.0 / 9.0;

        std::cout << "Temperature::Get() --> " << return_temperature <<
                " in " << scale << std::endl;

        return return_temperature;
}
// END IDL: Sensors::Temperature
```

supplierfilter.cpp

```
#include "orbsvcs/orbsvcs/Notify/Notify_StructuredPushConsumer.h"
#include "orbsvcs/orbsvcs/Notify/Notify_StructuredPushSupplier.h"

#include "ServerFilterSetUp.h"
```

```
#include "Filter.h"
#include <iostream>
#include <stdlib.h>

#define NOTIFY_FACTORY_NAME "NotifyEventChannelFactory"
#define NAMING_SERVICE_NAME "NameService"
#define CA_FILTER "Temperature  < 70"
#define SA_FILTER "Temperature > 10"
#define TCL_GRAMMER "TCL"
#define MAX_STRING_SIZE 256
#define NOTIFY_CHANNEL "myNotifyChannel"

extern long AirQuality;
extern long Temperature;
extern long WaterQuality;

class PushConsumer : public TAO_Notify_StructuredPushConsumer
{
private:
 char myname_[MAX_STRING_SIZE];


public:
 PushConsumer (const char* myname)
 {
          strcpy(myname_,myname);

 }

 virtual void push_structured_event (
    const CosNotification::StructuredEvent & notification,
    CORBA::Environment &/*ACE_TRY_ENV*/
    )
  ACE_THROW_SPEC ((
            CORBA::SystemException,
            CosEventComm::Disconnected
            ))
 {
  CORBA::Long val;
       CORBA::Long value;



  notification.remainder_of_body >>= val;

       std::cout << myname_ << " received event " << val << std::endl;
       for (int i=0; i<notification.filterable_data.length(); i++) {

                std::cout << "  " << (notification.filterable_data[i].name).in() << std::endl;

                if (notification.filterable_data[i].value >>= value)
                        std::cout << "  " << value << std::endl;

       }
 }
};
```

```
class PushSupplier : public TAO_Notify_StructuredPushSupplier
{
private:
char myname_[MAX_STRING_SIZE];

public:
 PushSupplier (const char* myname)
 {
        strcpy(myname_,myname);
 }

virtual void send_event (const CosNotification::StructuredEvent& event,
                 CORBA::Environment &ACE_TRY_ENV)
 {
//   ACE_DEBUG ((LM_DEBUG,
//             "%s is sending an event \n", myname_.fast_rep ()));
        std::cout << myname_ << " is sending an event" << std::endl;

   // add you own constraints here.
   TAO_Notify_StructuredPushSupplier::send_event (event,
                                  ACE_TRY_ENV);

 }
};


FilterClient::FilterClient (void)
{
 // No-Op.
 ifgop_ = CosNotifyChannelAdmin::OR_OP;
}

FilterClient::~FilterClient ()
{
 // No-Op.
}

void
FilterClient::init (int argc, char *argv [])
{
std::cout << "In server FilterClient::init, top" << std::endl;

 init_ORB (argc, argv);
std::cout << "In server FilterClient::init, after init_orb" << std::endl;

 resolve_naming_service ();
std::cout << "In server  FilterClient::init, after resolve_naming_service" << std::endl;

#if SUPPLIER
// Set up the Notify factory, and create the event channel
 resolve_Notify_factory ();
std::cout << "In server FilterClient::init, after resolve_Notify_factory" << std::endl;

 create_EC ();
std::cout << "In server FilterClient::init, after create_EC" << std::endl;
```

```
#endif

#if CONSUMER
 resolve_Notify_Channel();
 std::cout << "In server FilterClient::init, after resolve_Notify_Channel" << std::endl;
#endif



#if SUPPLIER
 create_supplieradmin ();
#endif

#if CONSUMER
 create_consumeradmin ();
#endif
std::cout << "In client FilterClient::init, after create_consumeradmin" << std::endl;

#if CONSUMER
 create_consumers ();
#endif
std::cout << "In client FilterClient::init, after create_consumers" << std::endl;

#if SUPPLIER
 create_suppliers ();
#endif

std::cout << "In client FilterClient::init, very bottom" << std::endl;


}

void
FilterClient::run ()
{

 send_events ();

}

void
FilterClient::init_ORB (int argc,
                char *argv [])
{
 this->orb_ = CORBA::ORB_init (argc,
                        argv,
                        "");

 CORBA::Object_ptr poa_object =
  this->orb_->resolve_initial_references("RootPOA");


 if (CORBA::is_nil (poa_object))
  {
//    ACE_ERROR ((LM_ERROR,
//                " (%P|%t) Unable to initialize the POA.\n"));
```

```
    return;
   }
  this->root_poa_ =
   PortableServer::POA::_narrow (poa_object);

  PortableServer::POAManager_var poa_manager =
   root_poa_->the_POAManager ();


  poa_manager->activate ();

}

void
FilterClient::resolve_naming_service ()
{
  CORBA::Object_var naming_obj =
   this->orb_->resolve_initial_references (NAMING_SERVICE_NAME);


  // Need to check return value for errors.
  if (CORBA::is_nil (naming_obj.in ()))
   //ACE_THROW (CORBA::UNKNOWN ());
         throw 0;

  this->naming_context_ =
   CosNaming::NamingContext::_narrow (naming_obj.in ());

}

void
FilterClient::resolve_Notify_factory ()
{
std::cout << "in FilterClient::resolve_Notify_factory, at top" << std::endl;

  CosNaming::Name name (1);
  name.length (1);
  name[0].id = CORBA::string_dup (NOTIFY_FACTORY_NAME);

std::cout << "in FilterClient::resolve_Notify_factory, just before resolve" << std::endl;
  CORBA::Object_var obj =
   this->naming_context_->resolve (name);

std::cout << "in FilterClient::resolve_Notify_factory, just before narrow" << std::endl;
  this->notify_factory_ =
   CosNotifyChannelAdmin::EventChannelFactory::_narrow (obj.in ());

}

void
FilterClient::create_EC ()
{
  CosNotifyChannelAdmin::ChannelID id;
  CosNaming::Name name (1);

  ec_ = notify_factory_->create_channel (initial_qos_,
```

```
                        initial_admin_,
                        id);

 // Register the Notification Channel with the Naming Service
 name.length (1);
 name[0].id = CORBA::string_dup (NOTIFY_CHANNEL);

 try {
        this->naming_context_->bind (name, ec_.in ());
 }
 catch (const CosNaming::NamingContext::AlreadyBound &) {
        std::cout << "Earlier server already bound the NOTIFY_CHANNEL" << std::endl;
        std::cout << "That's okay, just proceed using its value" << std::endl;

        // Have to resolve the notify channel, instead of using the new
        // event channel just created
        resolve_Notify_Channel();

 }


// ACE_ASSERT (!CORBA::is_nil (ec_.in ()));
 if (CORBA::is_nil (ec_.in())) {
        std::cout << "event channel is nil\n";
        throw 0;
 }
}

void FilterClient:: resolve_Notify_Channel()
{
        CosNaming::Name name;
        name.length(1);
        name[0].id = CORBA::string_dup(NOTIFY_CHANNEL);

std::cout << "Top of resolve_Notify_Channel" << std::endl;

        CORBA::Object_var notify_channel_obj = this->naming_context_->resolve(name);
std::cout << "resolve_Notify_Channel, after resolve " << std::endl;

        ec_ = CosNotifyChannelAdmin::EventChannel::_narrow(notify_channel_obj.in());
std::cout << "resolve_Notify_Channel, after narrow " << std::endl;

        if (CORBA::is_nil(ec_.in())) {
                std::cout << "Unable to find the notification channel" << std::endl;
        }

}

void
FilterClient::create_supplieradmin ()
{
 CosNotifyChannelAdmin::AdminID adminid = 0;

 supplier_admin_ =
  ec_->new_for_suppliers (this->ifgop_, adminid);
```

```
// ACE_ASSERT (!CORBA::is_nil (supplier_admin_.in ()));
 if (CORBA::is_nil(supplier_admin_.in())) {
        std::cout << "supplier_admin is nil" << std::endl;
        throw 0;
 }

 CosNotifyFilter::FilterFactory_var ffact =
  ec_->default_filter_factory ();


 // setup a filter at the consumer admin
 CosNotifyFilter::Filter_var sa_filter =
  ffact->create_filter (TCL_GRAMMER);

// ACE_ASSERT (!CORBA::is_nil (sa_filter.in ()));
 if (CORBA::is_nil(sa_filter.in())) {
        std::cout << "sa_filter is nil" << std::endl;
        throw 0;
 }

 CosNotifyFilter::ConstraintExpSeq constraint_list (1);
 constraint_list.length (1);

 constraint_list[0].event_types.length (0);
 constraint_list[0].constraint_expr = CORBA::string_dup (SA_FILTER);

 sa_filter->add_constraints (constraint_list);

 supplier_admin_->add_filter (sa_filter.in ());

}

void
FilterClient:: create_consumeradmin ()
{
 CosNotifyChannelAdmin::AdminID adminid = 0;

 consumer_admin_ =
  ec_->new_for_consumers (this->ifgop_, adminid);


// ACE_ASSERT (!CORBA::is_nil (consumer_admin_.in ()));
 if (CORBA::is_nil(consumer_admin_.in())) {
        std::cout << "consumer admin is nil" << std::endl;
        throw 0;
 }

 CosNotifyFilter::FilterFactory_var ffact =
  ec_->default_filter_factory ();

 // setup a filter at the consumer admin
 CosNotifyFilter::Filter_var ca_filter =
  ffact->create_filter (TCL_GRAMMER);

// ACE_ASSERT (!CORBA::is_nil (ca_filter.in ()));
```

```
 if (CORBA::is_nil(ca_filter.in())) {
          std::cout << "ca_filter is nil" << std::endl;
          throw 0;
 }

 /*  struct ConstraintExp {
          CosNotification::EventTypeSeq event_types;
          string constraint_expr;
          };
 */
 CosNotifyFilter::ConstraintExpSeq constraint_list (1);
 constraint_list.length (1);

 constraint_list[0].event_types.length (0);
 constraint_list[0].constraint_expr = CORBA::string_dup (CA_FILTER);

 ca_filter->add_constraints (constraint_list);


 consumer_admin_->add_filter (ca_filter.in ());

}

void
FilterClient::create_consumers ()
{

 // startup the first consumer.
//  ACE_NEW_THROW_EX (consumer_1,
//                PushConsumer ("consumer1"),
//                CORBA::NO_MEMORY ());
 do { try { consumer_1 = new PushConsumer ("consumer1") ; }
 catch (void *all) {
          std::cout << "Not enough memory" << std::endl;
 }
 } while (0);


ACE_TRY_NEW_ENV
 consumer_1->init (this->root_poa_.in (), ACE_TRY_ENV);

 consumer_1->connect (consumer_admin_.in (), ACE_TRY_ENV);
ACE_CATCH (CORBA::SystemException, se);
ACE_ENDTRY;


 // startup the second consumer.
//  ACE_NEW_THROW_EX (consumer_2,
//                PushConsumer ("consumer2"),
//                CORBA::NO_MEMORY ());
 do { try { consumer_2 = new PushConsumer ("consumer1") ; }
 catch (void *all) {
          std::cout << "Not enough memory" << std::endl;
 }
 } while (0);
```

```
ACE_TRY_NEW_ENV
 consumer_2->init (this->root_poa_.in (), ACE_TRY_ENV);

 consumer_2->connect (consumer_admin_.in (), ACE_TRY_ENV);
ACE_CATCH (CORBA::SystemException, se);
ACE_ENDTRY;

}

void
FilterClient::create_suppliers ()
{
 // startup the first supplier
//  ACE_NEW_THROW_EX (supplier_1,
//                    PushSupplier ("supplier1"),
//                    CORBA::NO_MEMORY ());
 do { try { supplier_1 = new PushSupplier("supplier1") ; }
 catch (void *all) {
          std::cout << "Not enough memory" << std::endl;
 }
 } while (0);


ACE_TRY_NEW_ENV
 supplier_1->init (this->root_poa_.in (), ACE_TRY_ENV);

 supplier_1->connect (supplier_admin_.in (), ACE_TRY_ENV);
ACE_CATCH (CORBA::SystemException, se);
ACE_ENDTRY;

 // startup the second supplier
//  ACE_NEW_THROW_EX (supplier_2,
//                    PushSupplier ("supplier2"),
//                    CORBA::NO_MEMORY ());
 do { try { supplier_2 = new PushSupplier("supplier2") ; }
 catch (void *all) {
          std::cout << "Not enough memory" << std::endl;
 }
 } while (0);


ACE_TRY_NEW_ENV
 supplier_2->init (this->root_poa_.in (), ACE_TRY_ENV);

 supplier_2->connect (supplier_admin_.in (), ACE_TRY_ENV);
ACE_CATCH (CORBA::SystemException, se);
ACE_ENDTRY;

}

void
FilterClient::send_events ()
{
 // operations:
 CosNotification::StructuredEvent event;
```

```
// EventHeader

// FixedEventHeader
// EventType
// string
event.header.fixed_header.event_type.domain_name = CORBA::string_dup("*");
// string
event.header.fixed_header.event_type.type_name = CORBA::string_dup("*");
// string
event.header.fixed_header.event_name = CORBA::string_dup("myevent");

// OptionalHeaderFields
// PropertySeq
// sequence<Property>: string name, any value
event.header.variable_header.length (1); // put nothing here

// FilterableEventBody
// PropertySeq
// sequence<Property>: string name, any value
event.filterable_data.length (3);
event.filterable_data[0].name = CORBA::string_dup("AirQuality");

event.filterable_data[1].name = CORBA::string_dup("Temperature");

event.filterable_data[2].name = CORBA::string_dup("WaterQuality");

for (int i = 0; i < 1000; i++)
  {
   event.filterable_data[0].value <<= (CORBA::Long)AirQuality;
         event.filterable_data[1].value <<= (CORBA::Long)Temperature;
         event.filterable_data[2].value <<= (CORBA::Long)WaterQuality;


         // any -- let remainder of the body send the event #
   event.remainder_of_body <<= (CORBA::Long)i;

ACE_TRY_NEW_ENV
   supplier_1->send_event (event, ACE_TRY_ENV);

   supplier_2->send_event (event, ACE_TRY_ENV);
ACE_CATCH (CORBA::SystemException, se);
ACE_ENDTRY;

  }

}
```

filter.h

```
#ifndef NOTIFY_FILTER_CLIENT_H
#define NOTIFY_FILTER_CLIENT_H

#include "orbsvcs/orbsvcs/CosNotifyChannelAdminC.h"
#include "orbsvcs/orbsvcs/CosNotifyCommC.h"
#include "orbsvcs/orbsvcs/CosNamingC.h"

class TAO_Notify_StructuredPushConsumer;
```

```
class TAO_Notify_StructuredPushSupplier;

class FilterClient
{
 // = TITLE
 //   Filter Client
 // = DESCRIPTION
 //   Client example that shows how to do Structured Event filtering
 //   in the Notification Service.

public:
 // = Initialization and Termination
 FilterClient (void);
 ~FilterClient ();

 void init (int argc, char *argv []);
 // Init the Client.

 void run ();
 // Run the demo.

protected:
 void init_ORB (int argc, char *argv []);
 // Initializes the ORB.

 void resolve_naming_service ();
 // Try to get hold of a running naming service.

 void resolve_Notify_factory ();
 // Try to resolve the Notify factory from the Naming service.

 void create_EC ();
 // Create an EC.

 void resolve_Notify_Channel();

 void create_supplieradmin();
 // Create the Supplier Admin.

 void create_consumeradmin ();
 // Create the Consumer Admin.

 void create_consumers ();
 // Create and initialize the consumers.

 void create_suppliers ();
 // create and initialize the suppliers.

 void send_events ();
 // send the events.

 // = Data Members
 PortableServer::POA_var root_poa_;
 // Reference to the root poa.

 CORBA::ORB_var orb_;
```

```
  // The ORB that we use.

  CosNaming::NamingContext_var naming_context_;
  // Handle to the name service.

  CosNotifyChannelAdmin::EventChannelFactory_var notify_factory_;
  // Channel factory.

  CosNotifyChannelAdmin::EventChannel_var ec_;
  // The one channel that we create using the factory.

  CosNotifyChannelAdmin::InterFilterGroupOperator ifgop_;
  // The group operator between admin-proxy's.

  CosNotification::QoSProperties initial_qos_;
  // Initial qos specified to the factory when creating the EC.

  CosNotification::AdminProperties initial_admin_;
  // Initial admin props specified to the factory when creating the EC.

  CosNotifyChannelAdmin::ConsumerAdmin_var consumer_admin_;
  // The consumer admin used by consumers.

  CosNotifyChannelAdmin::SupplierAdmin_var supplier_admin_;
  // The supplier admin used by suppliers.

  TAO_Notify_StructuredPushConsumer* consumer_1;
  // Consumer #1

  TAO_Notify_StructuredPushConsumer* consumer_2;
  // Consumer #2

  TAO_Notify_StructuredPushSupplier* supplier_1;
  // Supplier #1

  TAO_Notify_StructuredPushSupplier* supplier_2;
  // Supplier #2
};
#endif /* NOTIFY_FILTER_CLIENT_H */
```

serverfiltersetup.h

```
//#define CONSUMER 1
#define SUPPLIER 1
```

stock_i.h

```
#ifndef __SENSOR_PACKAGE_EXAMPLE_I_H
#define __SENSOR_PACKAGE_EXAMPLE_I_H

#include "QuoterS.h"
#include <string>
#include "TAO\orbsvcs\orbsvcs\CosEventCommC.h"
#include "TAO\orbsvcs\orbsvcs\CosEventChannelAdminC.h"

// IDL: interface AirQuality
class AirQuality_i : public POA_Sensors::AirQuality {
public:
```

```
  AirQuality_i();
  AirQuality_i(Sensors::Quality_t quality);

  // IDL: Values_t Get();
  Sensors::AirQuality::Values_t_slice * Get();
private:
  Sensors::Quality_t current;                    // current air quality
};

// IDL: interface WaterQuality
class WaterQuality_i : public POA_Sensors::WaterQuality {
public:
  WaterQuality_i();
  WaterQuality_i(Sensors::Quality_t quality);

  // IDL: Quality_t Get(in string quadrant)
  // IDL:          raises (Invalid_Quadrant);
  Sensors::Quality_t Get (const char * quadrant)
          throw (Sensors::WaterQuality::Invalid_Quadrant);

private:
  Sensors::Quality_t current;                    // current water quality
};

// IDL: interface Temperature
class Temperature_i : public POA_Sensors::Temperature {
public:
  Temperature_i();
  Temperature_i(Sensors::Temperature_t temperature);

  // IDL: void Configure(in TemperatureScale_t scale);
  void Configure(Sensors::TemperatureScale_t scale);

  // scale = 'F' or 'C'
  // IDL: Temperature_t Get(out char scale);
  Sensors::Temperature_t Get(CORBA::Char_out scale);

private:
  Sensors::Temperature_t current_temperature; // in Fahrenheit
  char current_scale;
  CosEventChannelAdmin::ProxyPushConsumer_var consumer;
};

#endif /* SENSOR_PACKAGE_EXAMPLE_I_H */
```

Appendix E
CORBA Notification Service Presentation

Event Service

- There are two serious limitations of the Event Channel defined by the OMG Event Service:
  - Event filtering is not supported
  - Different qualities of service (QoS) are not supported

Notification Service

- The primary goals of the CORBA Notification Service are to enhance the Event Service by providing filtering and QoS
- Clients of the Notification Service subscribe to specific events of interest by association filter objects with the proxies through which clients communicate with Event Channels
- With the Notification Service, each Channel, each Connection, and each Message can be configured to support desired QoS

Notification Service

- The Notification Service supports three types of events:
  - Untyped events – contained within an Any
  - Typed events – from Event Service
  - Structured events
    - New with Notification Service
    - Define a well known data structure to which many different types of events can be mapped, in order to support highly optimized event filtering

Notification Service

- Notification Service interfaces inherit directly from Event Service interfaces
- Notification Service interfaces are named similarly to corresponding event service interfaces
- Since the Notification Service version of a particular interface inherits from the Event Service equivalent of the same interface, an instance of the Notification Service can be widened to become an instance of the Event Service

Notification Service

- In addition to supporting multiple instances of each Proxy interface, each Notification Service event channel may also support multiple instances of the ConsumerAdmin and SupplierAdmin interfaces (defined in the CosNotifyChannelAdmin module)
- As with the Event Service, a typed version of the Notification Service event channel also exists that has similar architecture to the standard Notification Service event channel, with additional interfaces to handle typed communication

Notification Service

- The factory operations supported by a Notification Service interface through inheritance from the equivalent Event Service interface can be invoked to create a true Event Service version of a particular interface
- In the Notification Service specification, the issue of whether or not an instance of an Event Service style interface can be narrowed to an equivalent Notification Service style interface is left as an implementation detail

Notification Service

- Due to interface inheritance, an instance of an object supporting an instance of the Notification Channel (NCA::Event Channel) can be widened to one supporting the ECA::Event Channel interface, and henceforth be treated identically to the Event Service version of the event channel.
- The primary reason for this is to support backward compatibility for existing applications which use the Event Service

Notification Service Definitions

- ECA::EventChannel, NCA::EventChannel

- ECA::SupplierAdmin, NCA::SupplierAdmin
- ECA::ConsumerAdmin, NCA::ConsumerAdmin

- ECA::ProxyPushSupplier, NCA::ProxyPushSupplier
- ECA::ProxyPushConsumer, NCA::ProxyPushConsumer
- ECA::ProxyPullSupplier, NCA::ProxyPullSupplier
- ECA::ProxyPullConsumer, NCA::ProxyPullConsumer

Notification Service Definitions (cont'd)

- NCA::StructuredProxyPushSupplier
- NCA::StructuredProxyPushConsumer
- NCA::StructuredProxyPullSupplier
- NCA::StructuredProxyPullConsumer

- NCA::SequenceProxyPushSupplier
- NCA::SequenceProxyPushConsumer
- NCA::SequenceProxyPullSupplier
- NCA::SequenceProxyPullConsumer

Notification Service backward compatibility with the Event Service

- True Event Service clients (consumers and suppliers) can connect to the Notification Service using one of these techniques:

- Using the operations of the inherited ECA::EventChannel interface, instantiate the appropriate Event Service version of ConsumerAdmin or SupplierAdmin. Use the Admin interface to instantiate an event service style Proxy interface, then connect to that interface
- Obtain the appropriate Notification style Admin interface. Instantiate an event service style Proxy interface using inherited Event Service methods on the Admin interface. Connect to that interface.
- Use Notification style interfaces to instantiate the appropriate Proxy (ProxyPushConsumer, ProxyPullSupplier, etc.) Connect to that interface.

Notification Service backward compatibility with the Event Service

- Techniques 1 and 2 above result in a true event service client connected to a true event service style proxy interface associated with the channel.

- Technique 3 above results in an event service client being connected to a Notification Style proxy interface.
  - The Notification Style proxy interface is capable of filtering events, and can support QoS.
  - Thus, in this case an event service client can take advantage of the new Notification Service features

Notification Service Event Channel Factory

- The Event Channel Factory creates new instances of notification channels.
- At creation time, the client can specify varoius QoS and administrative properties that will be supported by the channel
- Standard administrative properties include:
  - Max # of events the channel will buffer at any one time (MaxQueueLength)
  - Max # consumers (MaxConsumers)
  - Max # suppliers (MaxSuppliers)

Notification Service Event Channel Factory

- Although the EventChannelFactory is the only interface in the Notification Service that is explicitly defined to be a factory:
  - The architecture of the Notification Service is hierarchical in nature
  - All objects defined as part of an event channel are created by parent objects
- For example:
  - ConsumerAdmin and SupplierAdmin instances are created by event channels
  - All ProxyObjects are created by an Admin interface

Notification Service Event Channel Factory

- In the Notification Service, all objects that create other objects assign a unique numeric identifier to the object (unique within the creating object)
- All objects that create other objects support an operation that returns the list of all unique identifiers of all objects created by that object
- All objects that create other objects also support an operation that, given a single unique identifier, returns the object reference of the creating object's child object

Notification Service Event Channel

- The Notification Service Event Channel (also called Notification Channel) supports the CosNotifyChannelAdmin::EventChannel interface

- The Notification Channel can support multiple Admin interfaces
- Each ConsumerAdmin and SupplierAdmin interface is a factory that creates the Proxy interfaces to which objects connect

- Admin objects can themselves have QoS and filter objects
- The Notification Service treats each Admin object as the manager of the Proxy objects it created

Notification Service Event Channel

- Each Notification Service Admin object has its own unique identifier within the Notification Service Event Channel
  - Each Notification Channel is also capable of creating Event Service style Admin interfaces, which do not have unique identifiers
- Upon creation, each Notification Channel instance initially supports a single ConsumerAdmin and SupplierAdmin instance
  - These are viewed as the default of that type object
  - Initially assigned an identifier value of 0

Notification Service Style Admin Objects

- The main idea underlying the support of multiple Admin objects per channel is to optimize the handling of clients with identical requirements
- Each instance of a Notification Service style Admin object is capable of creating and managing a set of Proxy objects
- The Notification Service identifies new interfaces for clients that send/receive events in the form of Structured Events or Sequences of Structured Events

Notification Service Style Admin Objects

- Message communication to and from the Notification Channel can be in terms of Any's, Structured Events, or Sequences of Structured Events

- The exact form of message communications supported by a Proxy object that was created by a Notification Services style Admin object is controlled by a flag provided as input to the operation on the Admin object when it created the Proxy

Notification Service Style Proxy Interfaces

- The Notification Service Admin objects can create:
  - Pure Event Service style Proxy objects
  - Notification Service style Proxy objects

Notification Service Style Proxy Interfaces

- Notification Service style Proxy objects can be subdivided into three categories:
  - Those that send and receive Anys
  - Those that send and receive Structured Events
  - Those that send and receive Sequences of Structured Events
- Note that both push and pull versions of each type of Proxy object are supported

Notification Service Style Proxy Interfaces

- Notification Service Proxy objects have two types of filters associated with them:
  - Forwarding Filters
    - Can be attached to all kinds of proxy objects
    - Constrain the events the objects will forward
  - Mapping Filters
    - Can only be attached to Supplier Proxy objects
    - Affect the priority or lifetime properties of each event received by a Supplier proxy object

Notification Service Style Proxy Interfaces

- A Proxy object with no associated filter objects defaults to forwarding all events it receives
- Filters can be associated either with the Proxy or with the Admin object (can have different filters on both)

Notification Service Style Proxy Interfaces

- The Notification Service supports well-defined translations of message formats in the case that an event is supplied in a format different than a particular consumer is designed to receive:
  - An Any can be converted into Structured Event format
  - An Any can be converted into Typed Event format
  - A Structured Event can be converted into an Any
  - A Structured Event can be converted into Typed Event format
  - A Typed Event can be converted into an Any
  - A Typed Event can be converted into a Structured Event

Notification Service Style Proxy Interfaces

- An invocation of the push_structured_events operation, which is supported by a SequencePushConsumer (with length n), is equivalent to n calls to the push_structured_event operation supported by a StructuredPushConsumer

Notification Service Style Proxy Interfaces

- The translation scheme raises the potential for an event to be wrapped multiple times as the result of multiple translations.
  - This is avoided by
    - having Proxy Consumers look down into the typecode of Anys to see if it corresponds to the typecode for a Structured Event
    - having consumers look down into the typecode of a Structured Event to see if it corresponds to an Any

Notification Service Style Proxy Interfaces

- One additional aspect of the Notification Service interfaces is that they support a means to share event subscription information between notification channels and their clients:
  - NotifySubscribe – supports an operation that allows each notification supplier to be notified when the set of events for which there are currently interested consumers changes
  - NotifyPublish -- supports an operation that allows each notification consumer to be notified when the set of event types which are currently being offered to the channel changes

- NotifySubscribe and NotifyPublish can be used together in order to optimize event communication by only transmitting events when necessary

Notification Service – Structured Events

- The OMG Event Service supports two types of events:
  - Untyped – Any
  - Typed – however, the typed interface has usually been found difficult to use
- To fix problems with typed events, the Notification Service introduced Structured Events
- Structured Events define a standard data structured into which a wide variety of event messages can be stored

Notification Service – Structured Events

Notification Service – Structured Events

- Each Structured Event consists of:
  - Header
    - Fixed Header
      - Domain name – identifies particular vertical industry domain in which the event type is defined (telecommunications, finance, etc.)
      - Type name – categorizes the event uniquely within the domain (CommunicationsAlarm, StockQuote, etc.)
      - Event Name – uniquely identifies the special instance of the event currently being transmitted
    - Variable Header – consists of zero or more value pairs ("ohf" = Optional Header Field), where each name is a string and each value is an Any. Contains per message QoS information
  - Body -- contains the contents of each event instance.
    - Filterable portion – ("fd" = Filterable Data), name/value pairs, where each name is a string and each value is an Any
    - Remainder of the body – defined as an Any (this can also be used as filterable data, although it is normally considered a separate field)

Notification Service – Structured Events – Standard Optional Header Fields (OHF)

- EventReliability, type is short, 0 = best effort, 1 = persistent
- Priority, type is short, -32,767 to +32,767, default=0
- StartTime, type is TimeBase::UtcT, absolute time after which the channel can deliver the event
- StopTime, type is TimeBase::UtcT, absolute time when the channel should discard the data
- Timeout, type is TimeBase::TimeT, relative time when the channel should discard the event

Notification Service – Structured Events – Standard Optional Header Fields (OHF)

- Priority and timeout properties may optionally be set within the header of a Structured Event

- The priority and timeout fields within the Structured Event override those set at the proxy level
- No object within the channel ever modifies the contents of a Structured Event
- It is envisoned that different vertical domains will define standard mappings of specific event types into Structured Events

Notification Service – Structured Events – Event Type Repository

- Structured Events are particularly useful when used together with an event type repository
- An event type repository completely describes the makeup of each time of event mapped into a structured event
- End users can use this meta-data to construct filters which subscribe to new instances of Structured Events that are dynamically added to the system

Notification Service – Event Filtering with Filter Objects

- Each Admin and Proxy interface defined by the Notification Service inherits the CosNotifyFilter::FilterAdmin interface, which supports operations for the maintenance of a list of filter objects
- Each Admin and Proxy object within a Notification Channel can have associated with it one or more filter objects
- These filter objects could be co-located in the same server process as the Notification Channel, or they can reside in their own address space (there is a communications penalty when matches are applied in a remote address space)

Notification Service – Event Filtering with Filter Objects

- There are two types of Filter Objects defined by the Notification Service:
  - CosNotifyFilter::Filter – affect forwarding decisions made by Proxy objects
  - CosNotifyFilter::MappingFilter – affect the way a Proxy object treats events with respect to QoS properties
- Each filter object encapsulates a set of one or more constraints specified in a particular constraint grammar

Notification Service – Event Filtering with Filter Objects

- Each constraint is a data structure consisting of two components:
  - A sequence of data structures, each of which indicates an event type
  - A string containing a boolean expression whose syntax conforms to some constraint grammar
- The sequence of event types in the constraint data structure contains the list of event types to which the particular constraint applies

Notification Service – Event Filtering with Filter Objects

- Many different constraint grammars may be supported by a particular implementation of the Notification Service
- However, every implementation is required to have the CosNotifyFilter::Filter grammar

Notification Service – Event Filtering with Filter Objects

- The convention is that an empty sequence of event type structures associated with a boolean constraint expression implies that the expression applies to all types of events

- A single element in the sequence of event type structures in which both fields are the empty string also implies that the expression applies to all types of events
- A type element whose value is the wildcard character {*,*} also indicates the boolean expression applies to all types of events

Notification Service – Event Filtering with Filter Objects

- Upon receipt of each event, each Proxy object invokes a "match" operation on each of its associated filter objects
- Match returns TRUE if any one of the constraints encapsulated in the filter object is satisfied by the event; FALSE otherwise
- If the match for all filter objects associated with the Proxy returns FALSE, the Proxy discards the event.
- Otherwise the event will be forwarded:
  - To all the proxy suppliers when the filtering was supplied by a proxy consumer, or
  - To the associated consumer when the filtering was performed by a proxy supplier

Notification Service – Event Filtering with Filter Objects

- The set of filter objects associated with an Admin object can only be modified by invoking operations on the Admin object itself
  - any such modifications affect all Proxy objects under the management of that Admin
- Filter objects can be added directly to an individual Proxy object by invoking the add_filter operation directly on the Proxy object itself

Notification Service – Event Filtering with Filter Objects

- Each Proxy Object can have two sets of filter objects associated with it:
  - Its own filters
  - Admin filters
- Upon creation of each Admin object, a flag will be set to say whether a proxy object ANDS or ORS the results of the 2 sets of filter objects
- Filter objects may be added directly to an individual Proxy Object by invoking the add_filter operation directly on the Proxy object itself

Notification Service – Event Filtering with Filter Objects

- A Proxy which has no filters associated with it will pass through all events it receives:
  - Proxy consumers will pass all events on to the Proxy suppliers
  - A proxy supplier will send all events on to its connected consumer

Notification Service – Event Filtering with Filter Objects

- CosNotifyFilter::Filter supports three styles of match operations:
  - Match – accepts Any as input
  - Match_structured – accepts Structured Events as input
  - Match_typed – accepts Typed Events as input

Notification Service – Event Filtering with Filter Objects

- CosNotifyFilter::Filter supports an attach_callback operation
  - The purpose of this operation is to associate with each filter object an interface upon which the subscription_change operation can be invoked, each time the set of constraints associated with the filter is modified
  - This feature is so event suppliers can be notified when the set of events subscribed to by the potential event consumers changes

Notification Service – Mapping Filter Objects

- There are many scenarios in which a consumer's opinion of an event's relative importance (priority and lifetime/expiration time) may differ from that of a supplier
- To enable consumers to affect priority and lifetime, the Notification Service uses Mapping Filter objects

Notification Service – Mapping Filter Objects

- Mapping Filter objects support the CosNotifyFilter::MappingFilter interface
- The main difference between this and the regular Filter interface is that mapping filters also associate a value with each constraint they encapsulate

Notification Service – Mapping Filter Objects

- In a Mapping Filter object, for the priority property, on a proxy supplier, after a match has been invoked:
  - Upon encountering the first constraint which the event satisfies, the match operation returns a result of TRUE and an output parameter is set to the value associated with the constraint
  - If the event does not satisfy any of the constraints associated with the Mapping Filter object, the match operation returns FALSE, and the default value associated with the Mapping Filter object will be returned as the output parameter
- Similarly for a Mapping Filter object, for a lifetime property

Notification Service – Mapping Filter Objects

- Upon return from the match operation, if the output parameter is FALSE, the proxy supplier will use the following rules to determine the priority that should be associated with the event:
  - If there is a priority property set in the event header, that value will be used
  - If there is no priority property set in the event header, but the event has inherited a priority property from a Proxy object with an associated priority property, that value will be used
  - Otherwise, the output parameter returned by the match operation will be used
- Similarly for a Mapping Filter object, for a lifetime property

Notification Service – Mapping Filter Objects

- Mapping Filter objects may be associated with:
  - Proxy suppliers
  - ConsumerAdmin interfaces
    - The mapping filters associated with a ConsumerAdmin object are shared by all proxy suppliers being managed by that ConsumerAdmin object
- Note that if a particular proxy supplier has a mapping filter, that overrides any mapping filter set for the same proxy on the ConsumerAdmin that manages that proxy supplier

Notification Service – Default Filter Constraint Language

- The default filter constraint language is the OMG Trading Service constraint language, with a few extensions:
  - Clarifications for some ambiguities in the Trader constraint language
  - Extensions allowing the components of complex data structures to be referenced
  - Some new features related to Notification Service needs

Notification Service – Default Filter Constraint Language

- Some examples of extensions:

- $ = current event as well as $name being a runtime variable
- A new symbol has been defined: <Component>, that implements an array, sequence, structure, or name value pair list of elements

Notification Service – Default Filter Constraint Language

- A <Component> is a collection of named identifiers.
  - However, multiple layers of encapsulation may not actually have identifier names associated with them
  - If the event type repository is in use, it may be able to supply the encapsulation information. Alternatively, when an event structure is pulled apart, unnamed layers will be passed over
  - Thus, the constraint author need not be concerned about unnamed layers of encapsulation

Notification Service – Default Filter Constraint Language

- Examples of event components:
  - Event.memA.Any.struct {int val, cnt;};
  - Event.memB.Any.Any.int;
  - Event.char;
  - Event.methA.(char key, Any.int types[10]);

- In 1., to reference cnt, $.memA.cnt
- In 2, to reference the unnamed integer, $.memB.
- In 3, to reference the data, $
- In 4, to reference types[3], $.methA.types[3]

Notification Service – Default Filter Constraint Language

- In general, arithmetic conversions follow C/C++ rules
- However, most arithmetic operations in the Notification Service are performed using CORBA::Long and CORBA::Double
- When a boolean is used in an arithmetic operation, it is treated as a CORBA::Long, with TRUE=1, FALSE=0

Notification Service – Default Filter Constraint Language

- When handed a constraint, the Notification Service can only guarantee that it is syntactically correct
  - It is only when events are actually filtered that it becomes possible to check that operands have valid data types
  - When invalid operands are encountered or when specified identifiers do not exist, "match" must immediately return FALSE

Notification Service – Default Filter Constraint Language

- Consider the following 4 events and the associated constraint:
- <$.a,'Hawaii'>,<$.c,5.0>
- <$.a,'H'>,<$.c,5.0>
- <$.a,5>, <$.c,5.0>
- <$.a,5>,<$.b,5.0>

- Constraint: ($.a+1>32) or ($.b==5) or ($.c>3)

Notification Service – Default Filter Constraint Language

- Any simple-types member of the fixed header or any property in the name/value pairs in the variable header and filterable data may be represented as runtime variables:
  - $.header.fixed_header.event_type.type_name=='CommunicationsAlarm' and $.header.fixed_header.event_name=='lost_packet' and $.header.variable_header(priority) < 2
  - Or, $type_name == 'CommunicationsAlarm' and $event_name=='lost_packet' and $priority < 2

Notification Service – Default Filter Constraint Language

- $domain_name is equivalent to $.header.fixed_header.event_type.domain_name
- type_name is equivalent to $.header.fixed_header.event_type.type_name
- event_name is equivalent to $.header.fixed_header.event_type.event_name

Notification Service – Default Filter Constraint Language

- Examples of Notification Service constraints:
  - Accept all CommunicationsAlarm events but no lost_packet messages:
    - $type_name=='CommunicationsAlarm' and not ($event_name=='lost_packet')
  - Accept only recent events (last 15 min. or so):
    - $origination_timestamp.high +2 < $curtime.high
  - Accept only Structured Events
    - $._repos_id == 'IDL:CosNotification/StructuredEvent:1.0'
  - Accept CommunicationsAlarm events with priorities ranging from 1 to 5
    - $type_name=='CommunicationsAlarm' and $priority >=1 and $priority <= 5

Notification Service – Quality of Service (QoS)

- The QoS model has the following components:
  - QoS Property Representation
  - Accessor operations for getting and setting QoS parameters at the:
    - Notification Channel
    - SupplierAdmins, Consumer Admins
    - Proxy suppliers and consumers
    - Individual event messages
  - QoS properties for notification
  - Negotiating QoS, including conflict resolution

Notification Service– QoS Property Representation

- <String,Any> pairs define QoS properties.
- CosNotification::PropertySeq represents property lists
- The Notification Service defines a number of standard property names, and defines the expected type and value range that should be contained in the associated Any

Notification Service– QoS Property Representation

- QoS can be set at various levels of scope by creating a QoSProperties sequence and selecting the interface for the particular scope:
  - get_qos, set_qos methods are available for:

- Notification channel
- Admin objects
- Individual proxy objects

- Also, for StructuredEvents, QoS properties can be set in the optional header field on a per-event basis

Notification Service– QoS Property Representation

- QoS importance:
  - Notification channel — lowest QoS importance
  - Proxy Admin — |
  - Proxy — V
  - Individual message — highest QoS importance

  - For example, QoS parameters in an individual message will override QoS settings in Proxy, Proxy Admin, and Notification Channel

  - In general, QoS Properties can be passed to a factory when creating a component.
  - The actual set of properties that is applied by a component is derived from merging the properties passed to the factory with the properties of the component at the higher scope level

Notification Service– QoS Properties

- Reliability
  - EventReliability
  - ConnectionReliability
- Priority
- Expiry Times
  - StopTime
  - Timeout
- Earliest Delivery Times
  - StartTime

Notification Service– QoS Properties (cont'd)

- Maximum Events Per Consumer
- Order Policy
  - AnyOrder
  - FifoOrder
  - PriorityOrder
  - DeadlineOrder
- Discard Policy
  - AnyOrder
  - FifoOrder
  - PriorityOrder
  - DeadlineOrder

Notification Service– QoS Properties (cont'd)

- Maximum Batch Size
- Pacing Interval

Notification Service– QoS Properties (cont'd)

Notification Service– Negotiating QoS and Conflict Resolution

- set_qos establishes QoS properties on its target object (Notification Channel, Admin, Proxy)

- get_qos returns the current QoS properties for its target object (Notification Channel, Admin, Proxy)
- validate_qos checks a potential QoS request to see if it would be supported, without actually changing the QoS settings
- validate_event_qos is similar to validate_qos except applies to QoS properties to be set in the header of a Structured Event

Notification Service– Negotiating QoS and Conflict Resolution

- UnsupportedQoS user exception is raised by certain operations to indicate that a QoS input parameter has an invalid or unsupported QoS

- BAD_QOS system exception can be raised to indicate that a QoS property in the header of a Structured Event is invalid or unsupported

Notification Service – Notification Channel Administrative Properties

- The administrative properties supported by the Notification Channel include:
  - MaxQueueLength
  - MaxConsumers
  - MaxSuppliers
  - RejectNewEvents
    - TRUE: pull style proxies will stop pulling from supplier when the total number of undelivered events=MaxQueueLength
    - FALSE: when the total number of undelivered events = MaxQueueLength, currently queued events will be discarded according to DiscardPolicy in order to make room for new events as they arrive

Notification Service – Notification Channel Subscription Sharing

- offer_change operation
  - Provided by NotifyPublish interface
  - Used by event suppliers to indicate to the channel the new types of events that they will supply, and the event types that they will no longer supply
- subscription_change operation
  - Provided by NotifySubscribe interface
  - A means of relaying subscription information, in the form of required event types, back to the source of events. It has two parameters:
    - One specifies event types that are required
    - One specifies event types that are no longer required

Appendix E
Real Time CORBA Standard

Real Time CORBA

- RT CORBA is an optional set of extensions to CORBA that allow CORBA ORBs to be used as part of a real time system

Real Time CORBA

- RT CORBA was proposed as an extension to CORBA
  - May 28, 1999 version was based on CORBA 2.2 plus the CORBA Messaging specification

Real Time CORBA – Schedulable Entity

- In the RT CORBA specification, the schedulable entity was threads provided by an underlying Operating System (OS)

Real Time CORBA -- Priority

- CORBA Priority:
  - May be associated with the current thread by setting the priority of the RTCORBA::Current object
  - Priority mapping interface maps the CORBA Priority to/from the native priority scheme of a given scheduler

Real Time CORBA -- Priority

- Priority Model policy:
  - Used to determine the priority at which a server handles requests from clients
  - Two models are supported:
    - CLIENT_PROPAGATED – server honors priority of the request sent by the client
    - SERVER_DECLARED – server handles request at a priority declared at the time of object creation

Real Time CORBA -- Priority

- In CLIENT_PROPAGATED model:
  - The client's CORBA priority is propagated in a new service context which is passed in the invocation request message
  - The server will map the CORBA priority ot its native RTOS priority and execute the invocation

Real Time CORBA – Preventing Priority Inversion

- Other tools to minimize priority inversion include:
  - A mutex interface

- Coordinates system resource usage
- Gives applications the same mutex interface/implementation as the ORB
- Policies for specifying and configuring communication protocols
- A threadpool on the server side

Real Time CORBA – Preventing Priority Inversion (cont'd)

- A policy to let the client set up multiple transport connections, each supporting a separate CORBA priority level
- A policy to specify non-multiplexed transport connections
- Priority transform on the server – allows implementation of various priority protocols

Real Time CORBA – Establishing a binding

- To set up a binding, RT CORBA uses the CORBA Messaging service's CORBA::Object::validate_connection operation
- This interface can be configured to:
  - Specify and configure the client-side protocol
  - Create priority bands
  - Set up a non-multiplexed connection between client and server

Real Time CORBA – POAs

- RT CORBA assumes the use of POAs
- Child POAs with particular real time policies can provide real time services
  - Non-RT portions of an ORB can run at the same time, and need less restriction on their behavior than is provided by the RT sections of the ORB

Real Time CORBA – Assumptions about Underlying Operating System

- An OS that implements IEEE POXIS 1003.1 real time extensions can provide end-to-end predictability; other OSes cannot
- The RT CORBA mutex protocol (priority inheritance or priority ceiling) is not specified
- The default priority protocol is not specified, since some operating systems will not provide that protocol

Real Time CORBA – Relationship of RT CORBA to CORBA Services

- Using the Concurrency Service, the Time Service, and the Event Service could impact end-to-end predictability
- Several elements of CORBA messaging are applicable to RT CORBA:

- Policy framework (levels at which policies can be applied or overridden)
- Mechanism for communicating server policies to clients
- Semantics of validate_connection are extended to provide explicit bind capability
- Invocation timeout in RT CORBA is the same as in CORBA Messaging

Real Time CORBA – Goals

- RT CORBA provides:
  - Resource management
  - Predictability
  - But also is less general purpose, more specific, than non-RT CORBA
- RT CORBA supports:
  - Hard real time
  - Soft real time
- RT CORBA only provides fixed priority scheduling
  - Dynamic scheduling is not currently provided

Real Time CORBA – Providing Real Time Support

- RT CORBA deals with invocations of IDL defined operations:
  - An operation invocation consists of a GIOP Request message followed by a GIOP Reply
- RT CORBA affects three phases of an activity:
  - "in transit" – a message within a transport protocol
  - "static" – a request held in memory
  - "active" – a thread scheduled to run on a processor

Real Time CORBA – Providing End to End Predictability

- One goal of RT CORBA is to provide end to end predictability, which includes:
  - Using thread priorities between client and server to resolve resource contention during invocations of operations
  - Limiting the duration of thread priority inversions
  - Limiting latency in operation invocations

Real Time CORBA – Providing End to End Predictability

- A RT CORBA ORB includes:
  - The scheduling mechanisms in the Operating System
  - The RT ORB itself
  - The transport protocol
  - The RT application software
- A RT CORBA system is dependent on the characteristics of the OS and the application, as well as the RT ORB

Real Time CORBA – Resource Management

- Resources come in three categories:
  - Processes
  - Storage
  - Communication resources

- RT CORBA offers control over:
  - Thread pools
  - Which objects the threads are used for, and what priorities they use
  - Some storage resources are appended to threadpools to handle concurrent requests after the number of threads provided has been used up

Real Time CORBA – Interoperability

- Instead of specifying a RT-IOP as an ESIOP, RT CORBA uses standard extension mechanisms provided by IIOP:
  - GIOP Service contexts
  - IOR Profiles
  - Tagged Components
- These allow IIOP to provide protocol support for RT CORBA

Real Time CORBA – Portability

- Porting an application between RT CORBA ORBs means that the application must be re-configured for the new ORB
- Non-RT CORBA applications may be ported to a RT ORB, but will not use the real time aspects of the RT ORB
- RT CORBA applications may be ported to a non-RT ORB, but the real time CORBA extensions will not be employed

Real Time CORBA – Architecture – RT CORBA Modules

- RTCORBA::Current
  - In the client, associates the CORBA priority with the current thread
- RTCORBA::Priority
  - Provides a universal, platform independent priority scheme, that allows consistent priority handling of invocations between CORBA systems with different priority schemes
- RT POA
  - Limits RT CORBA to part of the RT ORB
- RTCORBA::Threadpool
  - In the server, allows thread characteristics to be configured
- RT ORB
  - Handles RT ORB configuration, creation, and destruction
- RTCORBA::PriorityMapping
  - Maps RTCORBA Priority to/from native priorities.

Real Time CORBA – MUTEX Interface

- The RT CORBA::Mutex interface:
  - Implements some form of priority inheritance protocol. May include:
    - Simple priority inheritance
    - Priority ceiling protocol
- For consistency in priority inheritance, mutexes made available to the application must have the same priority inheritance properties as those used by the ORB for resource protection

Real Time CORBA – Threadpools

- In RT CORBA, ThreadPools are used to manage threads on the server. This includes:
  - Preallocation of threads
  - Thread partitioning

- A maximum limit on the number of threads a POA can use
- Additional requests can be buffered

Real Time CORBA – Priority Banded Connections

- To reduce priority inversions due to the use of a transport protocol that does not respect priorities, RT CORBA allows a client to communicate with a server via multiple connections:
  - Each connection handles invocations made at a different CORBA priority or range of CORBA priorities
- The selection of the connection is transparent to the application:
  - The application uses a single object reference as normal
- Also, RT CORBA allows a client to acquire a private (not shared) connection to a server

Real Time CORBA – Invocation Timeouts

- To improve predictability, RT CORBA applications may set a timeout on an invocation:
  - This limits the time a client is blocked waiting for a reply

Real Time CORBA – Configuration

- RT CORBA policies are provided to allow server side applications:
  - Configuration of threads on the server, through threadpools
  - Priority model selection
    - Propagated by client
    - Declared by server
- RT CORBA policies are provided to allow client-side applications:
  - To create sets of priority connections (banded connections) between clients and servers
  - To create a non-multiplexed connection to a server

Real Time CORBA – RT ORB

- RTCORBA::RTORB
  - An extension of CORBA::ORB
  - However, does not inherit from CORBA::ORB
- To obtain a reference to RTORB:
  - Call resolve_initial_references("RTORB");
- The reference to RTORB may not be passed as a parameter of an IDL operation
- The reference to RTORB may not be stringified

Real Time CORBA – RT ORB Initialization

- A RT ORB is initialized in CORBA::ORB_init()
- ORB_init for a RT ORB must handle:
  - -ORBRTpriorityrange <short>,<short>
  - The shorts are CORBA priorities for ORB internal threads

Real Time CORBA – Standard CORBA Exceptions with RT CORBA Minor Codes

- MARSHAL with minor code 2
  - Attempt to pass or stringify a RT ORB reference

- DATA_CONVERSION with minor code 1
  - RT ORB cannot map selected priorities (from –ORBRTPriority parameter to ORB_init) into native priority scheme
- INITIALIZE with minor code 1
  - Priority range set by –ORBRTPriority parameter was too narrow for the RT ORB to function properly
- BAD_INV_ORDER with minor code 1
  - Attempt to reassign priority
- NO_RESOURCES with minor code 1
  - No connection available that matches the priority of the request

Real Time CORBA – RT POA

- RT CORBA defines a RT POA, RTPortableServer::POA
  - PortableServer::POA instances may be treated as instances of RTPortableServer::POA
  - On a RT ORB, a call to resolve_initial_references("RootPOA") will return a RT POA
- A RT POA differs from a POA in 2 ways:
  - It provides additional operations and support for object level priority settings
  - It supports RT Policies

Real Time CORBA – Native Thread Priorities

- Base native thread priority – the native priority of a thread with which it was created, or to which it was explicitly set
- Active native thread priority – the priority of a thread considering its base native thread priority together with inherited priorities
- Priority inheritance protocol – used by a RT ORB in the implementation of threads and mutexes. May include (implementation issue):
  - Simple priority inheritance
  - Ceiling locking protocol
  - Etc.

Real Time CORBA – RT CORBA Priority

```
module RTCORBA {
   typedef short Priority;
   const Priority minPriority = 0;
   const Priority maxPriority = 32767;
};
```

Real Time CORBA – Priority Mapping

- For each RTOS in a system, the CORBA priority is mapped to the native thread priority scheme:
  - CORBA priority thus provides a common representation of priority across different RTOSes
  - A RTORB provides a default mapping for every RTOS it supports

Real Time CORBA – Priority Mapping

- Priority Mapping is defined as an IDL native type so that the priority mapping mechanism is exposed to the user.
- Language mappings for Priority Mapping are defined for C, C++, Ada, and Java

- Priority mapping is a programming language object, not a CORBA object:
  - Cannot use an object reference to access it

Real Time CORBA – Priority Mapping

```
namespace RTCORBA {
   classPriorityMapping {
        public:
        virtual CORBA::Boolean to_native ( RTCORBA::Priority          corba_priority,
   RTCORBA::NativePriority &native_priority );

        virtual CORBA::Boolean to_CORBA ( RTCORBA::NativePriority
        native_priority, RTCORBA::Priority &corba_priority);
   };
};
```

Real Time CORBA – Priority Mapping

- For efficiency, priority mapping operations do not raise any CORBA exceptions
  - Instead, they return a Boolean value indicating failure or success
- The RT ORB may make calls to_native and to_CORBA from different threads simultaneously (the implementations are re-entrant)

Real Time CORBA – RTCORBA::Current

- The RTCORBA::Current interface provides access to the CORBA priority of the current thread
  - An instance of CORBA::Current can be obtained by invoking resolve_initial_references("RTCurrent");
- A RT CORBA priority is associated with the current thread by setting the priority attribute of the RTCORBA::Current object
  - The priority is in the range 0 to 32767
  - Any values outside that range result in a BAD_PARAM exception

Real Time CORBA – CORBA Priority Models

- The RT CORBA Priority Models are:
  - Client Propagated
  - Server Declared
- The priority mode used is selected by use of the Priority Model policy

Real Time CORBA – CORBA Priority Models

- When the Server Declared model is selected:
  - The server_priority attribute indicates priority that will be assigned by default to CORBA objects managed by the POA
  - This can be overridden on a per-object basis

Real Time CORBA – CORBA Priority Models

- When the Client Propagated Model is selected:

- The server_priority attribute indicates the priority to be used for invocations from non-RT CORBA ORBs

Real Time CORBA – CORBA Priority Models

- An instance of the Priority Model policy is created with create_priority_mode
  - The attributes are initialized by create_priority_model policies
- The Priority Model Policy is applied to a RT POA at the time of its creation:
  - Either through an ORB level default
  - By including it in the policies parameter to create_POA

Real Time CORBA – CORBA Priority Models

- The Priority Model Policy is propagated so that the client ORB knows which Priority Model the target object is using:
  - This allows it to determine whether to send RT CORBA priority with object invocations on that object
    - In the case of priority banded connections, allows it to select the band connection
- The client may not override the Priority Model Policy itself
- If the client object supports CLIENT_PROPAGATED, the CORBA priority is carried with the invocation, and is used to ensure that all threads executing on behalf of the invocation run at the appropriate priority.
- The CORBA priority is also passed back from server to client, so it can be used to control the processing of the reply by the client

Real Time CORBA – CORBA Priority Models

- An object using the Server Declared priority model will publish its CORBA priority in its object reference:
  - When the object is invoked, the client can access the servant's priority. The client can use this information in conjunction with priority banded connections, for ex.
  - The client's RT CORBA priority is not passed with the invocation (request message)
  - The RT CORBA priority is not returned in a reply message
- The server priority assigned under the Server Declared priority model may be overridden on a per-object basis. This can be done at the time of object reference creation, or servant activation

Real Time CORBA – CORBA Priority Models

- Server priority setting routines include:
  - create_reference_with_priority
  - create_reference_with_id_and_priority
  - activate_object_with_priority
  - activate_object_with_id_and_priority

Real Time CORBA – Priority Transforms

- RT CORBA supports user-defined Priority Transforms
- Use of these allows priority models different from the Client Propagated or Server Declared
  - Priority Transforms are user-provided functions that map one RTCORBA::Priority value to another RTCORBA::Priority value
  - Priority Transforms modify the CORBA priority associated with an invocation during the processing of the invocation by a server

Real Time CORBA – Mutex Interface

- The mutex interface has the following operations:
  - lock()

- unlock()
- try_lock()

- The RT ORB interface has the following operations to support the Mutex interface:
  - create_mutex()
  - destroy_mutex()

Real Time CORBA – Mutex Interface

- A new RT CORBA Mutex object is obtained using RTCORBA::Manager::create_mutex()
- A Mutex object is unlocked by default.
- try_lock() works like lock() except that if it does not get the lock within the max_wait_time it returns false.
- The Mutex returned by create_mutex must have the same priority inheritance properties as those used by the ORB to protect resources.
- When a thread is executing in a protected region (protected by Mutex objects), it can be preempted only by threads that have a higher priority than the Mutex object.

Real Time CORBA – ThreadPools

```
struct ThreadPoolLane {
    Priority lane_priority;
    unsigned long static_threads;
    unsigned long dynamic_threads;
};

typedef sequence <ThreadPoolLane> ThreadpoolLanes;

const CORBA::PolicyType THREADPOOL_POLICY_TYPE;

interface ThreadpoolPolicy: CORBA::Policy {
    readonly attribute Threadpoolid threadpool;
};'
```

Real Time CORBA – ThreadPools

- RT CORBA Threadpool operations include:
  - create_threadpool_policy()
  - create_threadpool()
  - create_threadpool_with_lanes()
  - destroy_threadpool()
- Lanes are thread subsets at different assigned RT CORBA priority values
- When a threadpool is created, a ThreadpoolID is returned. This is later used to destroy the threadpool.
- The same threadpool may be associated with more than one POA. This is done by using the same ThreadpoolID in each call to POA_create()

Real Time CORBA – ThreadPools

- Threadpools without lanes must have configured:
  - Static threads -- The number of threads assigned to the threadpool
  - Dynamic threads -- The number of additional threads that may be created when the originally assigned threads are already in use, and a new request must be serviced
  - The default CORBA priority of the static threads. (Dynamic threads run at the priority of the request)

Real Time CORBA – ThreadPools

- Threadpools with lanes must have configured:

- Lane_priority – CORBA priority of all lanes in the current thread, including both the static threads and the dynamic threads belonging to the lane
- Static threads – threads allocated to the lane (as opposed to being allocated to the threadpool, as in threadpools without lanes)
- Dynamic threads – additional threads that may be added to a lane to handle a request.
- The allow_borrowing Boolean parameter must be configured to indicate whether the borrowing of threads by one lane from a lower priority lane is allowed

Real Time CORBA – ThreadPools

- A threadpool can be configured to buffer requests
  - When all threads are in use, and no more threads can be borrowed, then additional requests are buffered
  - Max_buffered_requests is the max number of requests that will be buffered by the threadpool
    - If set to zero, then the max number of requests is unbounded
  - Max_request_buffer_size is the max amount of memory that buffered requests may use
    - If set to zero, then the max amount of memory a buffered request may use is unbounded

Real Time CORBA – ThreadPools

- The Threadpool Policy can be applied at the POA or at the RT ORB
- A POA may only be associated with one threadpool, and with one threadpool policy
- A threadpool policy, when applied at the ORB, assigns the indicated Threadpool policy as the default threadpool policy for POA creation

Real Time CORBA – Implicit and Explicit Binding

- RT CORBA allows clients to control when a binding of an object reference occurs:
  - Uses CORBA::Object::validate_connection()
  - This is known as explicit binding

- With implicit binding (the default), binding occurs at an ORB-specific time, which can be as late as the first invocation on the object reference

Real Time CORBA – Priority Banded Connections

```
struct PriorityBand {
   Priority low;
   Priority high;
};
typedef sequence <PriorityBand> PriorityBands;

const CORBA::PolicyType PRIORITY_BANDED_CONNECTION;

interface PriorityBandedConnectionPolicy:CORBA::Policy {
   readonly attribute PriorityBands priority_bands;
};
```

Real Time CORBA – Priority Banded Connections

- An instance of the Priority Banded Connection policy is created with the create_priority_banded_connection_policy()
- The Priority Bands attribute may be assigned any number of priority bands
- Priority bands that cover a single priority may be mixed with those covering ranges of priorities

- No priority may be covered more than once
- All CORBA priorities need not be covered

Real Time CORBA – Priority Banded Connections

- The Priority Banded Connection Policy is applied on the client-side only, at the time of binding to a CORBA object
- However, the policy may be set by either the client or the server
- On the server, it may be set at the time of POA creation
  - Client exposed, propaged to client in the IOR

Real Time CORBA – Priority Banded Connections

- Whether bands are configured from the client to the server, the banded connection is always initiated from the client
- Once a priority band has been associated with a connection, it cannot be reconfigured udring the lifetime of the connection
- _bind_priority_band() is an implicit operation that is also called during an explicit bind, (validate_connection())

Real Time CORBA –Private Connections

- Create_private_connection_policy allows a client to obtain a private (not shared) connection to the server

Real Time CORBA – Invocation Timeout

- RT CORBA uses the Messaging::RelativeRoundtripTimeoutPolicy to set a timeout for receipt of a reply to an invocation

Real Time CORBA – Protocol Configuration

- RT CORBA allows 2 types of protocol configuration policies:
  - Server protocol policy
    - Used to select and configure communication protocols on the server side
    - Created with create_server_protocol_policy()
    - Allows any number of protocols to be specified
    - The order of protocols in the protocol list specifies preference
  - Client protocol policy
    - Used to select and configure communication protocols on the client side
    - Created with create_client_protocol_policy()
    - When applied to a bind, indicates protocols in orer of preference
    - May be set on either client side or server side

Real Time CORBA – RT CORBA Scheduling Service

- The RT CORBA scheduling service enforces fixed-priority RT scheduling policies across the RT CORBA system
- The sequence of events is as follows:
  - Start scheduling service running
  - Client obtains local reference to a client scheduler object
  - Calls schedule_activity() with a new deadline or priority, and a name
  - The scheduling service associates the name with the CORBA Priority (or deadline)

Real Time CORBA – RT CORBA Scheduling Service – Server scheduling code

```
ServerScheduler_var server_sched;

PortableServer::POA_var RTPOA = server_sched->create_POA
   (root_POA,"my_RT_POA",PortableServe::POAManager::_nil(), policies);

CORBA::Object_var obj1 = RTPOA ->create_reference("IDL:Object1:1.0");
CORBA::Object_var obj2 = RTPOA->create_reference("IDL:Object2:1.0");

server_sched->schedule_object(obj1,"Object1");
server_sched->schedule_object(obj2,"Object2");
```

Real Time CORBA – RT CORBA Scheduling Service – Server scheduling code

- In the code on the previous page, server_sched() creates a RT POA with a selection of RT policies – selected by the particular scheduling service implementation
- The calls to schedule_object set certain scheduling parameters for the objects

Real Time CORBA – RT CORBA Scheduling Service – Client scheduling code

```
clientScheduler_var client_sched;
object1_var obj1;
object1_var obj2;
//initialize obj1 and obj2
     ...

client_sched->schedule_activity("activity1");
obj1->method1();
obj2->method1();
client_sched->schedule_activity("activity2");
obj1->method2();
obj2->method2();
```

Real Time CORBA – RT CORBA Scheduling Service – Client scheduling code

- In the code on the previous page, method1() is called under the activity1 deadline, while method2() is called under the activity2 deadline
- The asssociation between the names "activity1" and "activity2" and their priorities was made prior to runtime
- Priorities of different activities are chosen based on the particular implementation of the scheduling service
- The scheduling service is the central place to change CORBA priorities
- Changes in priorities can be made without recompiling application code

Real Time CORBA – TAO Real Time CORBA Implementation, as of 9/18/2001

Real Time CORBA – TAO Real Time CORBA Implementation, as of 9/18/2001

- Unsupported:
  - POA Threadpool request buffering
  - POA Threadpool thread borrowing
  - Priority Transforms
  - ORBinit command line option

Real Time CORBA – TAO Real Time CORBA Implementation, as of 9/18/2001

- Future work:
  - Add persistent objects for RT POAs
  - Allow system wide purging of connection
  - All user-created threads are part of the default threadpool. They cannot be separated into lanes based on priorities.
  - Some CDR memory pools are global when they should be per-lane
  - RT CORBA implementation needs to be improved through benchmarking and performance testing
  - Need a queue_per_lane approach, where each threadpool lane owns a queue
  - Look at combining Real Time with fault tolerance

References

- "Real Time CORBA," OMG TC Document, ptc/99-05-03 May 28, 1999
- "CORBA Messaging Specification," orbos/98-05-05